EFFICIENT SEQUENTIAL DECISION-MAKING ALGORITHMS FOR CONTAINER INSPECTION OPERATIONS

BY SUSHIL MITTAL

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor David Madigan

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2008

ABSTRACT OF THE THESIS

EFFICIENT SEQUENTIAL DECISION-MAKING ALGORITHMS FOR CONTAINER INSPECTION OPERATIONS

by SUSHIL MITTAL

Thesis Director: Professor David Madigan

Sequential diagnosis is an old subject, but one that has become increasingly important recently. There exists a need for new models and algorithms as the traditional methods for making decisions sequentially do not scale. Motivated by the problem of container inspection at the U.S. ports, we investigate the problem of finding efficient algorithms for sequential diagnosis. More specifically, we formulate the port of entry inspection sequencing task as a problem of finding an optimal binary decision tree for an appropriate Boolean decision function. We provide new algorithms that are computationally more efficient than those previously presented by Stroud and Saeger [31] and Anand et al [1]. We achieve these efficiencies through a combination of specific numerical methods for finding optimal thresholds for sensor functions and two novel binary decision tree search algorithms that operate on a space of potentially acceptable binary decision trees. The improvements enable us to analyze substantially larger applications than was previously possible.

We try to solve the problem of finding an optimal inspection strategy by breaking it into two sub-problems - 1. Finding sensor threshold values that minimize the cost for a given binary decision tree and 2. "Searching" for the cheapest binary decision tree

in a large space of trees or equivalence classes of trees. For solving the first problem, we explore various standard non-linear optimization techniques and also propose a novel algorithm by combining the gradient descent method and Newton's method in optimization to compute optimal thresholds for any given tree. We propose two novel search algorithms - A stochastic search method and a genetic algorithms based search method, as a solution to the second sub-problem. We also propose "neighborhood" operations to move from one tree to another in the proposed tree space and prove that the tree space is irreducible under these neighborhood operations.

We report results from numerous experiments with and without imposing restrictions on the tree space and examine how the optimal binary decision trees vary with these changes. For example, for most of the work in this thesis, we restrict the tree space to constitute only "complete" and "monotonic" binary decision trees. Later, we "shrink" the tree space by discovering equivalence classes of trees while we "expand" the tree space by removing the monotonicity constraint.

Acknowledgements

I thank Dr. David Madigan for his guidance and support throughout my master's degree program. Working under him has been a really wonderful learning experience for me. I also owe him special thanks for helping me take some very important career decisions along the way.

I thank Dr. Fred Roberts for his constant support since the very first day of my work. His ideas and comments have greatly helped shape this work. I also thank him and DIMACS for supporting me financially during the last two years.

I thank Dr. Peter Meer for his ideas that constitute very important pieces of this work. I also thank him for his encouragement, support and for his unwavering belief in my abilities.

I wish to acknowledge Oncel Tuzel for his ideas and numerous detailed discussions on my work. I am also thankful to Dr. Endre Boros for helping me out with various problems through the long weekly discussions with him. Lastly, I thank my friends Saket Anand, Saumitr Pathak, Mahi Banday, Tashina Charagi and Raghu Makonahalli for providing me with enthusiasm, laughter and support throughout the progress of this work.

Dedication

To my Altec Lansing VS3151 speakers for their excellent sound in all the movies I watched during the last couple of years and the many more I could have watched if I weren't writing this thesis...

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Tables ix

List of Figures x

1. Introduction 1

1.1. Problem Definition 2

1.2. Preliminaries: The Stroud and Saeger [31] Approach 3

1.3. Our Approach 4

1.4. Previous Work 5

1.5. Organization of the Thesis 6

2. Cost of a Binary Decision Tree 7

2.1. The Cost Function 7

2.2. Sensor Model 8

2.3. Sensor Thresholds 9

2.4. Optimum Threshold Computation 10

2.4.1. Gradient Descent Method 11

2.4.2. Newton's Method 12

2.4.3. A Combined Method 13

2.5. Brief Summary 14

3. Completeness and Monotonicity 15

3.1. Complete and Monotonic Boolean Decision Functions 15
3.1.1. Complete BDFs 16
3.1.2. Monotonic BDFs 16
3.2. Complete and Monotonic Binary Decision Trees 17
3.2.1. Complete BDTs 17
3.2.2. Monotonic BDTs 17
3.3. A Few Trivial Proofs 18

4. Searching Through the Generalized Tree Space 21
4.1. Tree Neighborhood and Tree Space 21
4.1.1. Proof for the Tree Space Irreducibility for $n > 2$ 22
4.2. Tree Space Traversal 30
4.2.1. The Stochastic Search Method 30
4.2.2. Genetic Algorithms based Search Method 33

5. Shrinking the Tree Space 36
5.1. Tree Equivalence 36
5.1.1. Decision Equivalent BDTs 37
5.1.2. Cost Equivalent BDTs 37
5.2. Revisiting Monotonicity 40
5.3. Revisiting Completeness 45
5.4. Tree Reduction 47
5.4.1. Canonical Representation of an Equivalence Class 47
5.5. Space of Equivalence Classes of Irreducible Trees 48
5.5.1. Searching through the New Space 48

6. Experimental Results 51
6.1. Optimizing Thresholds 51
6.2. Searching the CM Tree Space 52
6.2.1. The Stochastic Search Method 53

6.2.2. Genetic Algorithms based Search Method 54
6.2.3. Going beyond 4 Sensors 54
6.3. Searching the Space of Equivalence Classes of Irreducible Trees 57
6.3.1. The Stochastic Search Method 57
6.3.2. Genetic Algorithms based Search Method 57

7. Conclusions and Future Research 59
7.1. Conclusions 59
7.2. Future Work 60
References 61

List of Tables

3.1. Number of complete and monotonic Boolean functions for 2, 3, 4 and 5 sensor types. 16

6.1. Summary of results for stochastic search for 4 sensor tree space. 53

6.2. Summary of results for genetic algorithm based search for 4 sensor tree space. 55

List of Figures

1.1. A binary decision tree τ with 3 sensors. The individual sensors classify good and bad containers towards left and right respectively. 4

2.1. A binary decision tree τ with 3 sensors. 7

2.2. Typical sensor model characterstics. 9

3.1. A Boolean function incomplete in sensor **a**, and the corresponding decision trees obtained from it. 18

3.2. A Boolean function non-monotonic in sensor **a**, and the corresponding decision trees obtained from it. 18

4.1. An example of notion of neighborhood. 23

4.2. An example of a start tree and a destination tree in τ^6. 24

4.3. A few examples of simple trees. 24

4.4. An example of a tree where removal of sensor **d** from the leftmost branch of the tree will result in a tree which is incomplete in sensor **b** (circled). 26

4.5. An example showing that any simple tree can be reached from any other simple tree using the four neighborhood operations repetitively. The first tree in the chain is σ_1 and the last one is σ_2. 30

4.6. An example showing that any arbitrary tree in τ^6 can be reached from any other arbitrary tree using the four neighborhood operations repeatedly. The sensor marked * in every tree is subject to a neighborhood operation while the circled sensors show a possible conflict with completeness constraint. 31

4.7. An example of the crossover operation. 34

5.1. An example of decision-equivalent binary decision trees. 38

5.2. An example of cost-equivalent binary decision trees. 39

5.3. An example of equivalence class of trees non-monotonic in sensor **a**. . . 41
5.4. An example of a one sensor non-monotonic tree and its corresponding monotonic tree. 42
5.5. Other possibilities of non-monotonic trees (i) and (ii), and their corresponding monotonic trees (iii) and (iv) respectively. 43
5.6. An example of a non-monotonic tree τ' that can be an optimum tree and its corresponding monotonic tree τ. Only one of the two instances of sensor **a** is monotonic in τ'. 44
5.7. An example of possible optimum threshold value of sensor **a**. 45
5.8. An example showing the procedure for obtaining unique canonical representation of a given tree. 49
6.1. Minimum costs for all 114 trees for 3 sensors. To avoid confusion, dashed vertical lines join markers for the same trees. 52
6.2. Best trees obtained for 4 sensors. Trees numbered 30995, 30959, 31011 and 31043 respectively. 54
6.3. Best trees obtained for 5 sensors over 100 runs. The cost of each of these trees is 41.4668. 56
6.4. Best trees obtained for 10 sensors over four different runs. Their costs are 8.6508, 8.5499, 8.7236 and 8.6189 respectively. 56

Chapter 1

Introduction

Making sequential decisions is an important problem that belongs to the otherwise broader area of decision theory. The key to the problem solving would be as simple as answering the question "What to do *next*?". Roughly speaking, it is a method of arriving at a final decision based on a *sequence* of smaller decisions. Making each of these decisions obviously involves answering a question. The choice of what question to ask is made on the basis of the answer to the previous question. However, there might be some problem-specific constraints that govern some of the rules about the sequencing of these questions. On the topmost level, there is usually an optimization function to make a choice of a specific question from a bunch of possible questions that can be asked at that point. The choice of this optimization function can vary based upon how the problem is formulated. For example, a user can choose one or more criteria to optimize and also the scope of optimization, i.e., greedy or global etc. Greedy optimization is usually simpler to achieve but doesn't always give the overall best solution to the problem. Therefore, in general, globally optimized systems are considered better systems because they are supposed to give a global (or near global) optimum solution. But obviously, finding a global solution is not always mathematically and computationally trivial. Throughout this thesis, we will try to address one such problem and discover various ways to solve it. We try to break up the problem into smaller, independent problems and propose one or more solutions to each one of them. Most of the proposed solutions are results of simple intuitions and basic concepts of optimization theory and combinatorics.

As explained in [1] and [29], sequential decision making problems appear in many areas. These include problems in the fields of communication networks for testing

connectivity in a circuit or paging celluar customers. In manufacturing, it is used for testing machines, fault diagnosis and routing customer service calls etc. In artificial intelligence and computer science, besides numerous other applications, it is used in particular for obtaining optimal derivation strategies in knowledge bases and coding decision tables etc. And finally, in medicine, it is used for diagnosing patients and sequencing their treatments. A selected list of references for such applications includes [15], [27] and [30].

Sequential diagnosis is an old subject, but one that has become increasingly important with the need for new models and algorithms as the traditional methods for making decisions sequentially do not scale. We investigate the problem to find algorithms for sequential diagnosis efficiently, through the container inspection procedure at the U.S. ports. Although the problem of container inspection is quite nicely formulated, we hope that the solutions we provide are generic to most sequential diagnosis applications.

1.1 Problem Definition

Finding ways to intercept illicit materials, in particular weapons, destined for the U.S. via the maritime transportation system is an exceedingly difficult task. Practical complications of inspection approaches involve the negative economic impacts of surveillance activities, errors and inconsistencies in available data on shipping and import terminal facilities, and the tradeoffs between costs and potential risks, among others. Until recently, even with increased budget and emphasis, and rapid development of modern technology, only a very small percentage of ships entering U.S. ports have their cargo inspected. Thus there is a great need to improve the efficiency of the current inspection processes. Recently, there has been a series of attempts to develop algorithms that will help us to inspect for and intercept chemical, biological, radiological, nuclear, and explosive agents, as well as other illicit materials. Such algorithms need to be developed with the constraint that seaports are critical gateways for the movement of international commerce. Slowing the flow long enough to inspect either all or a statistically significant random selection of imports would be economically intolerable [23].

As a stream of containers arrives at a port, a decision maker must decide which "inspections" to perform on each container. Current inspections include neutron/gamma emissions, radiograph images, induced fission tests, and checks of the ship's manifest. See [29], Section 2.5 and [4] for a more detailed description about the types of sensors that can be used. The specific sequence of inspection results will ultimately result in a decision to let the container pass through the port, or a decision to subject the container to a complete unpacking. Finding algorithms for sequential diagnosis that minimize the total "cost" of the inspection procedure, including the cost of false positives and false negatives, presents serious computational challenges that stand in the way of practical implementation.

1.2 Preliminaries: The Stroud and Saeger [31] Approach

We will think in the abstract of containers having "attributes" and having a sensor to test for each attribute; we will use the terms attribute and sensor interchangeably. In practice, we dichotomize attributes and represent their values as either 0 ("absent" or "ok") or 1 ("present" or "suspicious"), and we can think of a container as corresponding to a binary attribute string such as 011001. Classification then corresponds to a binary decision function F, that assigns each binary string to a final decision category. If the category must be 0 or 1, as we shall assume, F is a *Boolean decision function (BDF)*. Stroud and Saeger [31] consider the problem of finding an optimal *binary decision tree (BDT)* for calculating F. In the BDT, the interior nodes correspond to sensors and the leaf nodes correspond to decision categories. Two arcs exit from each sensor node, labeled left and right. By convention, the left arc corresponds to a sensor outcome of 0 and the right arc corresponds to a sensor outcome of 1. Figure 1.1 provides an example of a binary decision tree with three sensors denoted **a**, **b**, and **c**[1]. Thus, for example, if sensor **a** returns a zero ("ok"), sensor **b** returns a one ("suspicious"), and sensor **c** returns a one ("suspicious"), the tree outputs a one (i.e., a conclusion that something is wrong with the container).

[1]We allow duplicates of each type of sensor. Thus, we allow multiple copies of a sensor (of type **a**, and similarly for **b** and **c**). When we speak of n sensors, we mean n types and allow such duplicates.

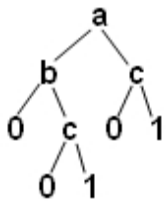


Figure 1.1: A binary decision tree τ with 3 sensors. The individual sensors classify good and bad containers towards left and right respectively.

Even if the Boolean function F is fixed, the problem of finding an "optimal" BDT for it is hard (NP-complete) [21]. Brute force enumeration can provide a solution. However, even if the number of attributes, n, is as small as 4, this is not practical. In present-day practice at busy US ports, we understand that n is of the order of 3 to 5, but this number is likely to grow as sensor technology becomes more advanced. Even under special assumptions, Stroud and Saeger [31] were unable to produce feasible methods for finding optimal binary decision trees beyond the case $n = 4$. They ranked all trees with up to 4 sensors according to increasing tree costs using a measure of cost we describe in Chapter 3, Section 2.1.

1.3 Our Approach

Following work of Stroud and Saeger [31], we report on new algorithms that are more efficient computationally than those presented by Stroud and Saeger [31]. We break the overall problem into two sub-problems. In the first problem we try to determine the optimum thresholds of the sensors at inspection stations. We describe efficient approaches to the computation of sensor thresholds (Chapter 3, Section 2.3), that seek to minimize the total cost of inspection. The second problem deals with the determination of the optimum sequence of inspection or the structure of the inspection decision tree in order to achieve the minimum expected inspection cost. This problem is similar to finding optimal sequential inspection procedure for reliability systems as described by [7], [19], [20], [3], [14], [13], and [2]. We also modify the special assumptions of Stroud and Saeger [31] to allow search through a larger number of possible Boolean decision functions, and introduce an algorithm for searching through the space of allowable

binary decision trees that avoids searching through the Boolean decision functions entirely. Later we modify the space again by putting better theoretical constraints on it. However, all our experiments parallel those of Stroud and Saeger [31].

1.4 Previous Work

The problem of inspecting containers is quite well-posed by now. Many people have worked on the problem, and all of them assume different models, different model parameters and they try to analyze the problem by converting it to into various different kinds of optimization problems. For example, [6] uses a polyhedral description of decision trees and develop a large scale linear programming model for container inspection. It incorporates various realistic limitations like budget, sensor capacity and time limits etc. Also, it assumes a very generic sensor model in a sense that it allows multiple thresholds for each sensor. Finally, it also allows a mixture of various inspection strategies instead of single best strategies. In [34], the optimization method is illustrated for inspection systems with decision functions of series, parallel, series-parallel, and parallel-series Boolean functions. For a comprehensive description of all these methods, please see [5].

In terms of the model parameters and other assumptions, the work that we present in this thesis is somewhat similar to the one presented in [1] and [29]. Both in [1] and [29], the authors report on an experimental analysis of the robustness of the conclusions of the Stroud and Saeger [31] analysis and show that the optimal inspection strategy (or the optimum BDT) is remarkably insensitive to variations in the parameters needed to apply the Stroud and Saeger [31] method. In their first set of experiments, they vary the values of costs of false negative and false positive and the prior probability of the occurence of a bad container in the ranges given by Stroud and Saeger [31] in [31]. But throughout these experiments, they use fixed threshold values (set such that the probabilities of false negative and positive are equal) for all the sensors across all the BDTs. In their second set of experiments, they fix the values of misclassification costs and the prior probability of a bad container and then find the optimum cost of any given BDT over the range of sensor threshold values by an exhaustive search with a fixed step

size. Both these sets of experiments were performed exhaustively over all possible trees with 3 and 4 sensors, beyond which it becomes infeasible to find the optimum strategy using exhaustive search. As we shall see in the chapters to follow, the work presented in this thesis tries to achieve targets similar to the ones in their second set of experiments.

1.5 Organization of the Thesis

The thesis is organised as follows. Chapter 2 explains the sensor model including the relevance of the sensor threshold, cost of a binary decision tree and efficient ways to compute it. Chapter 3 discusses the theoretical constraints of the Boolean decision functions being "complete" and "monotonic", given by Stroud and Saeger [31]. We modify these constraints to suit binary decision trees instead of Boolean functions by defining complete and monotonic BDTs. Chapter 4 explains various algorithms to "search" for the optimum trees in a space of complete and monotonic BDTs. In Chapter 5, we remodify the definitions of complete and monotonic BDTs by extending them to "equivalence classes" of complete and monotonic trees. We also define the "irreducibility" of a BDT and argue that only irreducibile trees can belong to the class of most optimum trees. Later, we modify the previously discussed search algorithms to suit the newer space of trees. The results from Chapters 3, 4 and 5 are combinedly presented in Chapter 6. In Chapter 7 we give conclusions and future work.

Chapter 2

Cost of a Binary Decision Tree

Following Anand et al. [1] and Stroud and Saeger [31], we assume the cost of a binary decision tree comprises two components: (i) the cost of utilization of the tree and (ii) the cost of misclassification of the tree. In general, the cost of utilization of each individual sensor has several smaller costs associated with it: the unit cost of inspecting one item with it, the fixed cost of purchasing and deploying it and the delay cost from queuing up at the sensor station. In our study, we have disregarded the fixed and delay costs and take only the unit cost of inspection into account.

2.1 The Cost Function

Keeping above things in mind, the cost of utilization of a tree is computed by performing a summation over the cost of each sensor in the tree times the probability that a container is inspected by that particular sensor. We compute the cost of misclassification for a tree by adding the probabilities of false positive and false negative misclassifications by the tree and multiplying by their respective costs. Both the costs described above depend on the distribution of the containers (into good and bad) and the probabilities of misclassification of the individual sensors.

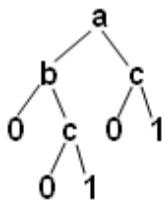


Figure 2.1: A binary decision tree τ with 3 sensors.

For example, consider the decision tree τ in Figure 2.1 with 3 sensors. The overall cost

function to be optimized can be written as:

$$\begin{aligned} f(\tau) &= P_0\left(C_{\mathbf{a}} + P_{\mathbf{a}=0|0}C_{\mathbf{b}} + P_{\mathbf{a}=0|0}P_{\mathbf{b}=1|0}C_{\mathbf{c}} + P_{\mathbf{a}=1|0}C_{\mathbf{c}}\right) \\ &\quad + P_1\left(C_{\mathbf{a}} + P_{\mathbf{a}=0|1}C_{\mathbf{b}} + P_{\mathbf{a}=0|1}P_{\mathbf{b}=1|1}C_{\mathbf{c}} + P_{\mathbf{a}=1|1}C_{\mathbf{c}}\right) \\ &\quad + P_0\left(P_{\mathbf{a}=0|0}P_{\mathbf{b}=1|0}P_{\mathbf{c}=1|0} + P_{\mathbf{a}=1|0}P_{\mathbf{c}=1|0}\right)C_{FP} \\ &\quad + P_1\left(P_{\mathbf{a}=0|1}P_{\mathbf{b}=0|1} + P_{\mathbf{a}=0|1}P_{\mathbf{b}=1|1}P_{\mathbf{c}=0|1} + P_{\mathbf{a}=1|1}P_{\mathbf{c}=0|1}\right)C_{FN} \end{aligned} \quad (2.1)$$

Here, P_0 and P_1 are the prior probabilities of occurrence of "good" (ok or 0) and "bad" (suspicious or 1) containers, respectively (so $P_0 + P_1 = 1$). For any sensor s, $P_{s=i|j}$ represents the conditional probability that the sensor returns an output i given that the container is in state in j, $i, j \in \{0, 1\}$. For real-valued attributes, Stroud and Saeger [31] describe a simple Gaussian model, which, combined with a specific threshold, leads to the requisite conditional probabilities; we discuss this further below. C_s is cost of utilization of sensor s, and C_{FP} and C_{FN} are the costs of a false positive and a false negative respectively. (The notation here differs from that in [1]). In the above expression, the first and second terms on the right hand side of equation 2.1, together give the total cost of utilization of the tree τ while the third and fourth terms represent the costs of positive and negative misclassifications respectively.

2.2 Sensor Model

We use the same sensor model that was proposed by Stroud and Saeger in [31] and was also used in [1] and [29]. It is basically a threshold model using counts (e.g., Gamma radiation counts). If the count exceeds some threshold, we conclude that the attribute being tested for is present. We assume that the sensor readings for good containers follow a Gaussian distribution and so do the readings for bad containers. Figure 2.2 below shows a typical sensor model assumed for a sensor s.

K_s is the called the discrimination power of the sensor and is basically the separation between the means of the two Gaussians. Each of the two Gaussians can have a different spread which also varies for various sensor types. The thin red line represents a hard

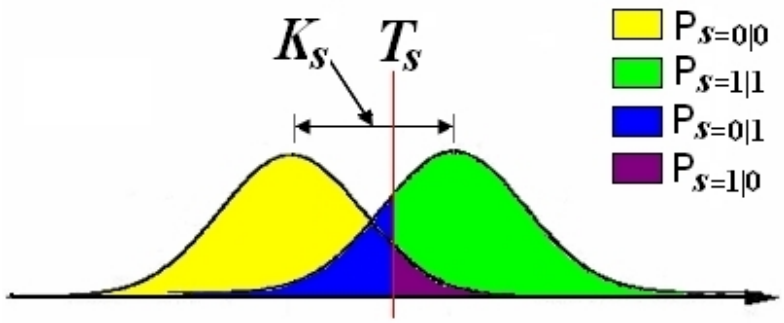


Figure 2.2: Typical sensor model characterstics.

threshold T_s. If we assume that all the containers with readings below T_s are labelled bad and all the containers with readings above T_s are labelled good, then as shown in the figure, the yellow region would represent the probability of correct detection of good containers for sensor s. Similarly, the green region would represent the probability of correct detection of bad containers for s. Likewise, the blue and the purple regions respectively give the probabilities of false negative and false positive for s[1]. Therefore, trivially,

$$P_{s=0|0} + P_{s=1|0} = 1 \tag{2.2}$$

$$P_{s=1|1} + P_{s=0|1} = 1 \tag{2.3}$$

2.3 Sensor Thresholds

As mentioned in the previous section, every sensor s is associated with a hard threshold T_s. The variation of sensor thresholds obviously impacts the overall cost of the tree. While sensor characteristics are a function of design and environmental conditions, the thresholds can, at least in principle, be set by the decision maker. However, throughout our work, we restrict multiple sensors of the same type in a tree to have identical thresholds. Although this assumption is quite realistic, even in the absence of such assumption the optimization techniques discussed further in this chapter should still,

[1]A part of the green region overlaps with the purple one. Basically, the entire region inside the second Gaussian on the right side of the threshold should be considered green. Similarly, the entire region inside the first Gaussian to the left of the threshold should be considered yellow.

in principle, be applicable. Mathematically, for any given tree τ, a set of optimum thresholds can be defined as a vector of threshold values that minimizes the overall cost function $f(\tau)$ for that tree. Anand et. al. [1] describe the outcomes of experiments in which individual sensor thresholds are incremented in fixed-size steps in an exhaustive search for optimal threshold values, and trees of minimum cost are identified. For example, for number of sensors types, $n = 4$, [1] reported $194,481$ experiments leading to lowest cost trees, with the results being quite similar to those obtained in experiments in [31]. Unfortunately, the methods do not scale and quickly become infeasible as the number of sensor types increases.

2.4 Optimum Threshold Computation

One of the aims of our work is to calculate the optimum sensor thresholds for a tree more efficiently and avoid an exhaustive search over a large number of threshold values for every sensor. The exhaustive search method suffers from a lot of drawbacks like a large search step size and limited range of search. Apart from this, the exhaustive search algorithm grows exponentially in computational time with the number of sensors, hence making it practically infeasible to go beyond a very small number of sensors. To deal with these drawbacks, we implemented various standard algorithms for nonlinear optimization problems. We note that the objective function, $f(\tau)$ is expected to be multimodal with respect to the various sensor thresholds. We used random restarts to address this concern. In the next two sub-sections we describe a couple of standard algorithms for solving non-linear optimization problems being applied to our problem. We also explain how these algorithms suffer from some limitations. To address these limitations, in the third sub-section, we come up with a novel, combined method for obtaining the optimum thresholds. For more details about these methods please see [24].

2.4.1 Gradient Descent Method

Gradient descent method or steepest descent method is one of the most famous techniques to solve optimization problems. To find a local minimum of a function using gradient descent, we takes steps proportional to the negative of the gradient of the function at the current point. In this method we form a vector of thresholds by randomly picking a threshold value for each sensor within some fixed range. Further, we find the partial differentials of the total cost function $f(\tau)$, (e.g., see Equation 2.1), with respect to each sensor threshold T_s, and form their vector $\partial\mathbf{f}$ by evaluating each of those partial differentials at the threshold values selected above. Therefore,

$$\partial\mathbf{f} = \left[\frac{\partial f}{\partial T_{\mathbf{a}}} \; \frac{\partial f}{\partial T_{\mathbf{b}}} \cdots \frac{\partial f}{\partial T_{s_n}}\right]^{\mathrm{T}}. \tag{2.4}$$

The threshold vector is then updated using the equation:

$$\mathbf{T} = \mathbf{T} - \lambda\partial\mathbf{f} \tag{2.5}$$

where λ is a very small, fixed step size. By doing this iteratively, we perform a gradient descent on the overall cost function, $f(\tau)$ towards its minimum. The following algorithm describes this method.

Algorithm 1 Gradient Descent Method for Optimum Threshold Computation

```
1  Initialize T_start as a vector of random threshold values
2  T ← inf
3  while |T − T_start| > 0.001 of T_start, do
4      T ← T_start
5      Compute ∂f
6      T_start ← T_start − λ∂f
7  end while
8  Output T_opt ← T
```

As we can see, algorithm depends mainly on the choice of the step size λ. The method is quite effective at limiting the exponential growth of computation with increasing number of sensors. Also, it usually gives a minimum lower than the exhaustive search method due to much finer resolution in step size. For our experiments with 3

and 4 sensor trees, $\lambda = 10^{-4}$ gave fairly good results with convergence achieved in a few hundred iterations. However, it suffers from the usual drawback of choosing the step size heuristically. If the step size is very small, the algorithm might take a long time to converge to the optimum solution. If it is too big, we may never hit the minimum at all.

2.4.2 Newton's Method

To eliminate the problem of setting the value of λ heuristically, we search for the minimum cost by using Newton's optimization method. In this method, the constant step size λ is replaced by the inverse of the Hessian matrix $\mathbf{H}f(\tau)$. The Hessian matrix is a square matrix of second order partial derivatives of the overall cost function $f(\tau)$. Since all the second derivatives of $f(\tau)$ are continuous over the sensor thresholds, the Hessian matrix for our problem is symmetric and is given by:

$$\mathbf{H}f(\tau) = \begin{bmatrix} \frac{\partial^2 f}{\partial T_{\mathbf{a}}^2} & \frac{\partial^2 f}{\partial T_{\mathbf{a}} \partial T_{\mathbf{b}}} & \cdots & \frac{\partial^2 f}{\partial T_{\mathbf{a}} \partial T_{s_n}} \\ \frac{\partial^2 f}{\partial T_{\mathbf{b}} \partial T_{\mathbf{a}}} & \frac{\partial^2 f}{\partial T_{\mathbf{b}}^2} & \cdots & \frac{\partial^2 f}{\partial T_{\mathbf{b}} \partial T_{s_n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial T_{s_n} \partial T_{\mathbf{a}}} & \frac{\partial^2 f}{\partial T_{s_n} \partial T_{\mathbf{b}}} & \cdots & \frac{\partial^2 f}{\partial T_{s_n}^2} \end{bmatrix} \tag{2.6}$$

Other than that, as shown following, the algorithm is quite similar to the gradient descent method.

Algorithm 2 Newton's Method for Optimum Threshold Computation

```
1  Initialize T_start as a vector of random threshold values
2  T ← inf
3  while |T − T_start| > 0.001 of T_start, do
4      T ← T_start
5      Compute ∂f
6      T_start ← T_start − [Hf(τ)]^{-1} ∂f
7  end while
8  Output T_opt ← T
```

Though the computation of the Hessian matrix is a little expensive and tedious, the method quickly converges in fewer iterations than the gradient descent method. The

convergence of this method depends largely on the starting vector $\mathbf{T_{start}}$. Since an absolute prior knowledge of the neighborhood of the minimum is absent, this method occasionally drifts in the wrong direction and hence fails to converge.

2.4.3 A Combined Method

For the above algorithm to converge to a minimum, it is required that the Hessian matrix $\mathbf{H}f(\tau)$ is a positive definite and well-conditioned matrix. But in practice, it might not be the case. Therefore, we explored alternative approaches to computing positive definite approximations to $\mathbf{H}f(\tau)$. These methods involve modified Cholesky decomposition schemes and have been nicely summarized by [16]. For example, a naive way to convert a non-positive definite matrix into a positive definite matrix is to decompose it to $\mathbf{LDL^T}$ (where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{D}$ is a diagonal matrix) form and then make all the non-positive elements of $\mathbf{D}$ positive. This crude approximation may result in the failure of factorization of the new matrix or make it very different from the original matrix. Therefore to address this issue more reasonably, we use a modified $\mathbf{LDL^T}$ factorization method from [18] which incorporates small error terms in both $\mathbf{L}$ and $\mathbf{D}$ at every step of factorization. Further, if the Hessian matrix $\mathbf{H}f(\tau)$ is ill-conditioned, we take small steps towards the minimum using the gradient descent method until it becomes well-conditioned. In this way we try to combine the advantages of both gradient descent and Newton's method. The following algorithm summarizes the final scheme for finding the optimum thresholds.

Algorithm 3 A Combined Method for Optimum Threshold Computation

```
1   Initialize T_start as a vector of random threshold values
2   T ← inf
3   while |T − T_start| > 0.001 of T_start, do
4       T ← T_start
5       Compute ∂f
6       Compute Hf(τ)
7       if Hf(τ) is not positive definite, then
8           Make Hf(τ) positive definite
9       end if
10      if Hf(τ) is well-conditioned, then
11          T_start ← T_start − [Hf(τ)]^{-1} ∂f
12      else
```

13 $\quad\quad\quad \mathbf{T_{start}} \leftarrow \mathbf{T_{start}} - \lambda\partial\mathbf{f}$
14 $\quad\quad$ **end if**
15 **end while**
16 Output $\mathbf{T_{opt}} \leftarrow \mathbf{T}$

2.5 Brief Summary

As we saw in this chapter, these optimization techniques provide better ways to converge to the set of optimum thresholds for each sensor. Using these techniques, the problem of calculating optimum thresholds only increases linearly with the increasing number of sensor types as opposed to its exponential growth using the exhaustive search method (in a fixed range and a fixed step size). Also, these optimization techniques work on *any* given binary decision tree irrespective of any theoretical limitation(s) on their structure, which we impose in the next chapter.

Chapter 3

Completeness and Monotonicity

In this chapter, we discuss some of the theoretical restrictions applied on the structure of binary decision trees. We will also prove how these restrictions suit our sensor model. The total number of binary decision trees that can be generated for a given number of sensor types increase very rapidly with the increasing number of sensor types. For n sensor types, the number of Boolean Decision Trees (BDTs) is given recursively by $N_n = 2 + nN_{n-1}^2$. For more details on this please see [31]. Table 3.1, Column 2 shows the total number of BDTs for $n = 2, 3, 4$, and 5. It becomes practically infeasible even to enumerate all trees for $n = 5$. In the following sections we first explain the restrictions applied by Stroud and Saeger [31] on Boolean decision functions (BDFs). Then, due to various reasons, we extend these restriction beyond Boolean functions to BDTs themselves.

3.1 Complete and Monotonic Boolean Decision Functions

According to Stroud and Saeger [31], BDTs representing only a "complete" and "monotonic" Boolean function, correspond to the potential best (cheapest) trees. Their claim, although quite intuitive, lacks a proof. But, they are able to get a significant amount of reduction in the number of "feasible" Boolean functions (see [31]) and hence the total number of BDTs. Table 3.1, shows some of the numbers for $n = 2, 3, 4$, and 5. Column 2 shows the total of number of BDTs possible, while Column 3 shows the number of complete and monotonic (CM) Boolean functions and Column 4 shows the total number of possible BDTs that correspond to these CM Boolean functions.

Number of Sensor Types	Number of Distinct BDTs	Number of CM BDFs	Number of BDTs from CM BDFs
2	74	2	4
3	16,430	9	60
4	1,079,779,602	114	11808
5	5 x 10^{18}	6894	263,515,920

Table 3.1: Number of complete and monotonic Boolean functions for 2, 3, 4 and 5 sensor types.

3.1.1 Complete BDFs

A Boolean decision function is called *complete* if all attributes contibute towards the output. However, a more formal definition could be stated by defining "incomplete" trees as follows.

Incomplete BDFs. A Boolean decision function F over n attributes will be called *incomplete* if F can be calculated by finding at most $n-1$ attributes and knowing the value of the input string on those attributes. The function F will be called *complete* if all the attributes contribute towards the output.

Therefore, if we list out the function output for all different combinations of input attribues, then for every attribute, switching the attribute value (from 0 to 1 or vice-versa) should flip the output at least once.

3.1.2 Monotonic BDFs

The definition of monotonic Boolean functions is a little trickier than that of complete Boolean functions.

Monotonic BDFs. Given two strings of input attributes, $x_1, x_2, ..., x_n$ & $y_1, y_2, ..., y_n$ such that $x_i > y_i \quad \forall\, i \in \{1, n\}$, a Boolean decision function F over these attributes will be called *monotonic* if and only if $F(x_1, x_2, ...x_n) > F(y_1, y_2, ..., y_n)$. In simpler words, if $F(x_1, x_2, ...x_n) = 1$ and $F(y_1, y_2, ..., y_n) = 0$ (given, $x_i > y_i \quad \forall\, i \in \{1, n\}$), then F is monotonic.

3.2 Complete and Monotonic Binary Decision Trees

We propose here definitions of monotonicity and completeness for BDTs themselves rather than limiting them to just the Boolean functions whence the trees are derived. We do this because of two reasons. Firstly, unlike Boolean functions, binary decision trees may not always consider all individual sensor outputs to give a final classification and hence incomplete and/or non-monotonic Booleans functions can in fact lead to useful trees. Therefore, it seems more natural to apply restrictions on the structure of BDTs and not the Boolean functions. Secondly, as we describe in Chapter 4, we apply various techniques to "search" for the optimum trees, instead of enumerating all of them and sometimes enlarging the "space" of objects to search can lead to more efficient search algorithms.

3.2.1 Complete BDTs

Consider, as an example, the Boolean function and its corresponding BDT's in Figure 3.1. The Boolean function is incomplete since the function does not depend on the attribute **a**. However, trees (i) and (ii), while representing the incomplete function faithfully, are themselves potentially viable trees with no redundancies present. Trees (iii) and (iv) on the other hand, are problematic insofar as they each contain identical subtrees at **a**. Sensor **a** is redundant in tree (iii) and tree (iv). Such considerations lead to the following definition of complete BDT.

Complete BDTs: A binary decision tree will be called *complete* if every sensor (attribute) occurs at least once in the tree and, at any non-leaf node in the tree, its left and right sub-trees are not identical.

3.2.2 Monotonic BDTs

Next consider the Boolean function and its corresponding BDTs in Figure 3.2. The Boolean function is not monotonic - when $\mathbf{b} = 1$ and $\mathbf{c} = 0$, then $\mathbf{a} = 0$ yields an output of 1 whereas $\mathbf{a} = 1$ yields an output of 0. Except for tree (i), the corresponding trees also exhibit this non-monotonicity because in all of those trees, there is a right

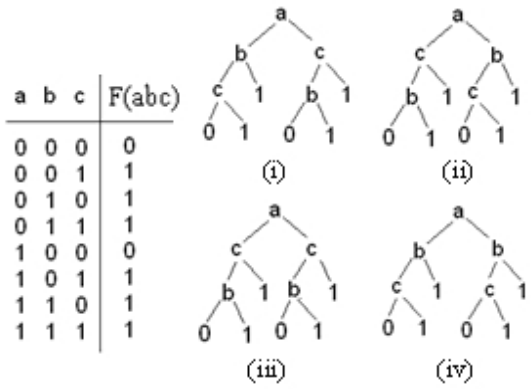


Figure 3.1: A Boolean function incomplete in sensor **a**, and the corresponding decision trees obtained from it.

arc from **a** to 0 or a left arc from **a** to 1 or both. However, tree (i) has no problems and might well be a useful tree. Thus, we have the following definition of monotonic BDTs.

Monotonic BDTs. A binary decision tree will be called *monotonic* if all leaf nodes emanating from a left branch are labeled 0 and all leaf nodes emanating from a right branch are labeled 1.

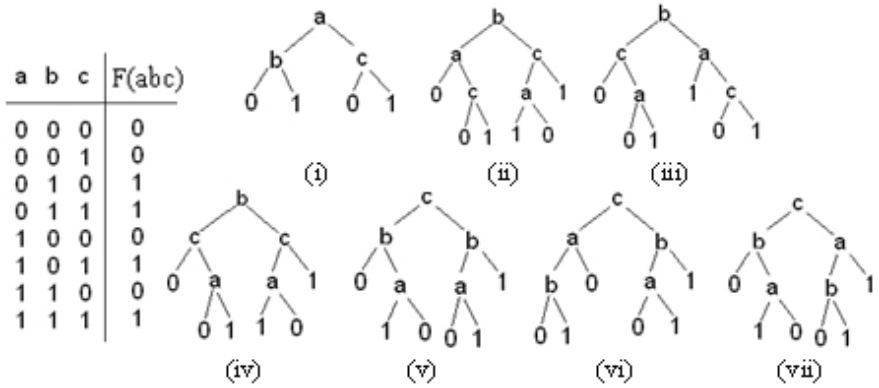


Figure 3.2: A Boolean function non-monotonic in sensor **a**, and the corresponding decision trees obtained from it.

3.3 A Few Trivial Proofs

In this section we prove some of the things related to complete and monotonic BDTs. We claim to "expand" the space of BDTs derived from complete and monotonic Boolean

functions to complete and monotonic trees. Therefore, we prove that all the trees that were initially there in Stroud and Saeger's [31] analysis are still present in our space of trees. We also justify the completeness and monotonicity constraints by showing that an incomplete or non-monotonic tree cannot potentially be a best tree.

Theorem 3.1. *All binary decision trees corresponding to complete Boolean functions are complete.*

Proof: Let F be a complete Boolean function over n attributes. Then, according to the definition of complete Boolean function, its output (at least once) should depend on every attribute. That means that for any attribute s, there must exist a set of input attribute values in which switching only the value of s (either from 0 to 1 or vice-versa) should change the output of the Boolean function. That clearly implies that in a BDT obtained from this Boolean function, there must be at least one instance of s where its ouputs to left and right correspond to complementary overall outputs. Therefore, it means that the presence of attribute s in that tree is not redundant and hence the tree is complete in attribute s. A similar reasonig proves that the tree is complete in all other attributes.

Theorem 3.2. *All binary decision trees corresponding to monotonic Boolean functions are monotonic.*

Proof: We prove this theorem by contradiction. Let F be a monotonic Boolean function over n attributes. Let τ be a non-monotonic BDT, non-monotonic in attribute s with F as its Boolean function. Now, if that is the case, then somewhere in the tree, one of the following must exist: (i) there is a 1 to the left of s, (ii) there is a 0 to the right of s or (iii) there is a 1 to the left of s and a 0 to the right of s. Now, in case (i) there must be a 0 leaf somewhere down in the right subtree of s and similarly, in case (ii) there must a 1 leaf somewhere down in the left subtree of s. Therefore, in all the three cases, there must be (at least one) set of attribute values, in which switching the value of only attribute s from 0 to 1, would switch the overall output of the tree from 1 to

0. In terms of Boolean function, this would mean that there exist a set of attribute values wherein switching just the value of attribute s from 0 to 1 changes the output of Boolean function from 1 to 0. Clearly, this would imply that the Boolean function is non-monotonic in attribute s which is contradictory to what we assumed about F.

Chapter 4

Searching Through the Generalized Tree Space

By generalizing the idea of completeness and monotonicity to BDTs, in the newer space of trees, the number of trees increase even more rapidly than in the space of trees from complete and monotonic Boolean functions. For example, for $n = 3$ we now have 114 complete and monotonic trees instead of 60 trees from complete and monotonic Boolean functions. Similarly, for $n = 4$, we now have 66,600 trees as compared to 11,808 trees earlier and for $n = 5$, we now have more than 22.5 billion trees as compared to 263,515,920 trees earlier. Therefore, it becomes practically infeasible to enumerate all trees and check each one's cost beyond $n = 4$. Therefore, in this chapter we introduce a notion of "neighborhood" of the trees in the tree space. We then propose to "search" for the optimum trees using the algorithms described in the later sections of the chapter.

4.1 Tree Neighborhood and Tree Space

Expanding the space of trees in which to search for a cost-minimizing tree to the space of complete, monotonic trees, "CM tree space" can be beneficial. While finding a cost-minimizing tree in CM tree space also presents a significant computational challenge as the number of sensors increases, we are able to address this challenge via heuristic search strategies that build on notions of neighborhoods in this space. Chipman et al. [9] and Miglio and Soffritti [26] provide a comparison of various definitions of neighborhood and proximity between trees. Also, Chipman et al. [8] describe methods to traverse the tree space and in what follows we develop a similar approach. We define neighbors in CM tree space via the following four kinds of operations on a tree.

- *Split*: Pick a leaf node and replace it with a sensor that is not already present in that branch, and then insert arcs from that sensor to 0 and to 1.

- *Swap*: Pick a non-leaf node in the tree and swap it with its parent node such that the new tree is still monotonic and complete and no sensor occurs more than once in any branch.

- *Merge*: Pick a parent node of two leaf nodes and make it a leaf node by collapsing the two leaf nodes below it, or pick a parent node with one leaf node, collapse both of them and shift the sub-tree up in the tree by one level. In both the operations the resultant tree should still be complete and monotonic.

- *Replace*: Pick a node with a sensor occurring more than once in the tree and replace it with any other sensor such that no sensor occurs more than once in any branch.

Figure 4.1 shows an example of neighboring trees obtained from these operations for a particular tree. Please note that the average number of neighbors for a tree increases with the number of sensor types.

4.1.1 Proof for the Tree Space Irreducibility for $n > 2$

Let τ^n represent the entire space of CM trees in n sensors. We will prove that any tree τ_2 in τ^n can be obtained from any other tree τ_1 in τ^n by an arbitrary sequence of neighborhood operations (split, swap, merge, and replace). First, we will define the notion of a simple tree. Then we will show that given τ_1, we can get from this tree to some simple tree σ_1 by these neighborhood operations. Similarly, we can get from τ_2 to some other simple tree σ_2. Next we show that any simple tree can be reached from any other simple tree by these operations and therefore we can reach σ_2 starting from σ_1. It is easy to see that the entire process of getting from an arbitrary tree to a simple tree is exactly reversible. For example, any split operation can be reversed using a merge operation and since, as we will see, we only merge nodes with both children as leaves, the converse is also true. The swap and replace operations can be reversed by opposite swap and replace operations, respectively. Thus, we see that we can get from σ_2 to τ_2 using the steps to reach τ_2 from σ_2 in the exact reverse order. Therefore, putting the three pieces together, we can go from τ_1 to σ_1 to σ_2 to τ_2. However, this proof is only

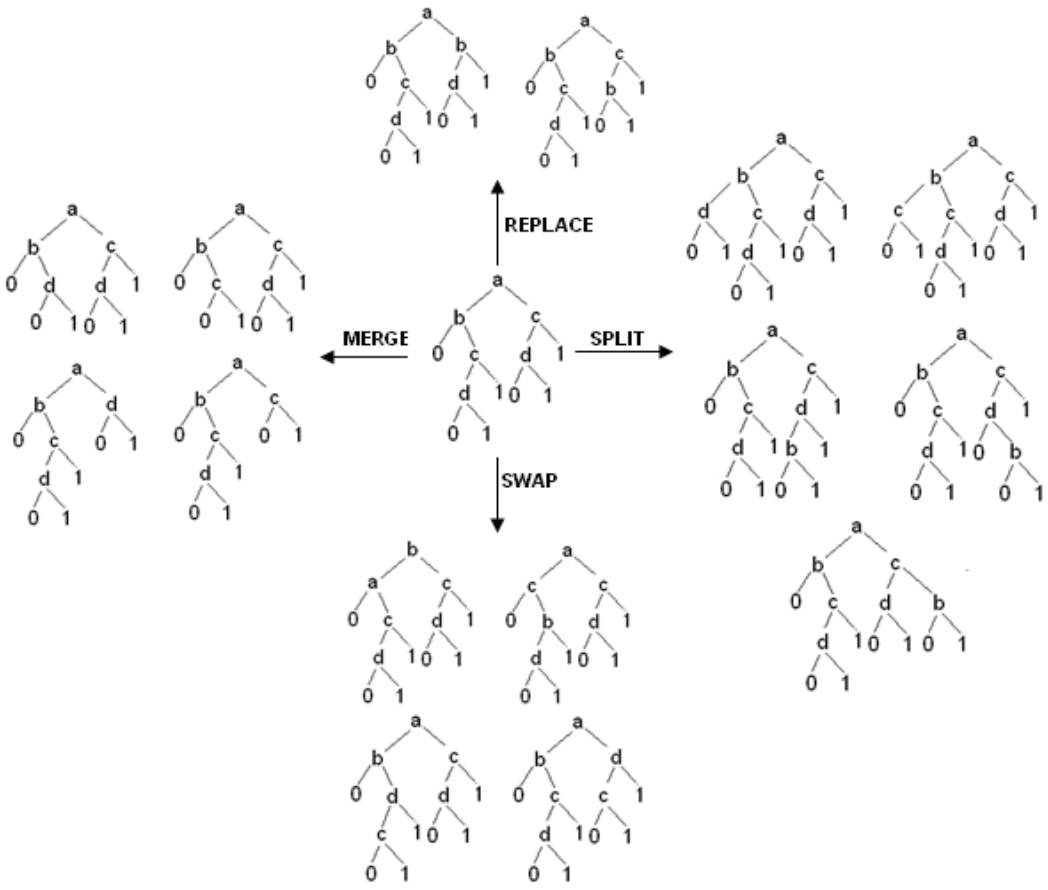


Figure 4.1: An example of notion of neighborhood.

valid for $n > 2$. For $n = 1$, the tree space consists of just one tree and hence the notion of tree neighborhood does not exist. For $n = 2$, there are just four trees in the space and the space is not irreducible under the neighborhood operations. For convenience in understanding, we will use one single example (Figure 4.6) that covers all the three pieces in τ^6. Figure 4.2 shows the starting tree τ_1 and the destination tree τ_2 used in the example.

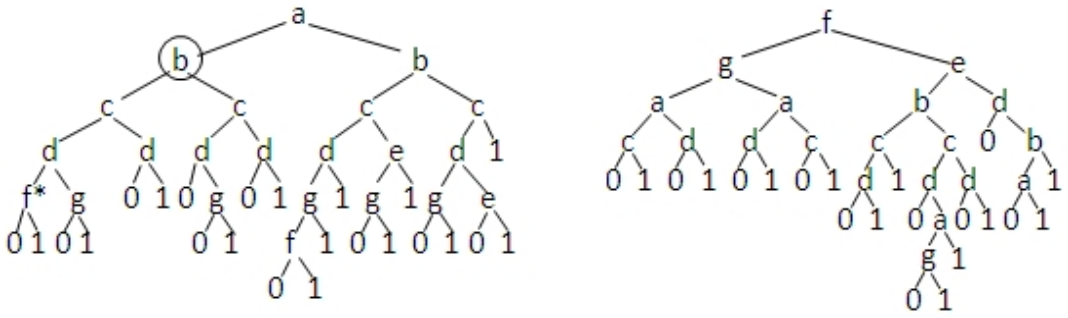


Figure 4.2: An example of a start tree and a destination tree in τ^6.

Simple tree. A simple tree is defined as a complete and monotonic decision tree in which every sensor occurs exactly once in such a way that there is exactly one path (called the "essential path") in the resultant tree with all sensors in it. For example, Figure 4.3 shows a few examples of simple trees for $n = 4$.

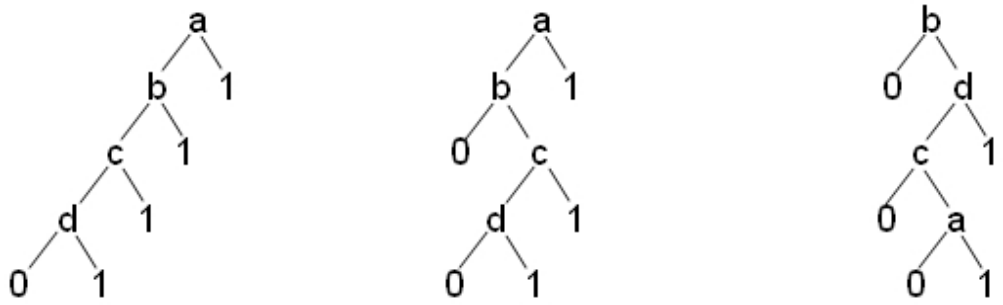


Figure 4.3: A few examples of simple trees.

We prove that we can reach some simple tree from any arbitrary tree. While we prove that, we will make frequent use of an algorithm called *"smartMerge"* which is used to merge (in the sense of repeated neighborhood operations) the left or right subtree of a given tree, in such a way that after every step in this algorithm, the resultant tree is complete and monotonic, i.e., every intermediate tree belongs to τ^n. First we describe

the types of problems associated with subtree merger and then we describe the details of the *smartMerge* algorithm. Later, we will describe how to use this algorithm to arrive at a simple tree from any arbitrary tree by iterative removal of subtrees at different depths in the tree.

The repeated merger of nodes to remove a subtree in a tree is non-trivial since we cannot merge a node that would lead to the overall tree becoming incomplete at a certain node. If we remove a sensor of depth k with both the child nodes as leaves, there is at most one of $k-2$ nodes (starting from the $k-2$ depth node in the same path, upwards till the root node) where the resultant tree can become incomplete. In other words, there can be at most one of $k-2$ nodes in the resultant tree, whose left subtree would become exactly identical to the right subtree after the removal, thus resulting in an incomplete tree which may not belong to τ^n. This is so, because after we remove the sensor at depth k and insert a leaf there, the tree cannot become incomplete at that leaf. Also, since we always insert an appropriate leaf (0 if the sensor is a left node, 1 otherwise) after removing a sensor, the tree cannot become incomplete at the parent node of the new leaf. Also, we can merge a sensor only if that sensor is present at some other place(s) in the tree, so that the resultant tree is still complete in n sensors after the merger. Therefore, if we merge such a sensor, the presence of another instance of that sensor at some other place in the tree guarantees that there is one of the $k-2$ nodes where the tree cannot become incomplete after the merger. Now we can deduce that there can be at most one out of $k-3$ nodes in the tree where a tree can become incomplete. However, in general, we still need to inspect all $k-2$ nodes for completeness. Figure 4.4 shows an example of a tree where removal of the sensor **d** from the leftmost branch of the tree results in the tree becoming incomplete in a higher sensor **b** (circled). Notice that since **d** is also present at other places in the tree, the tree cannot become incomplete in sensor **a** (circled) after we remove sensor **d**.

smartMerge Algorithm. Let τ represent a tree in τ^n and let τ_{sub1} and τ_{sub2} be the two subtrees at its root node. Further, let us assume that we want to merge τ_{sub1} and retain τ_{sub2}. Let the maximum depth of τ_{sub1} be k (as measured from the root node in τ). We then propose to merge the deepest node n_{1k} in τ_{sub1} first. As argued

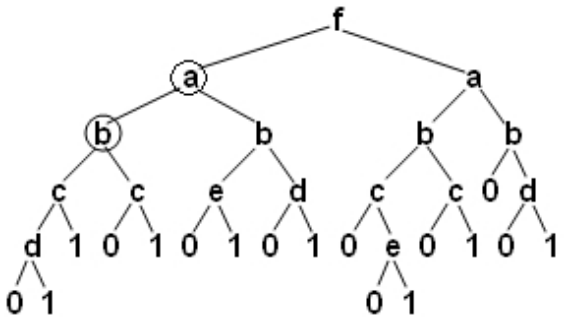


Figure 4.4: An example of a tree where removal of sensor **d** from the leftmost branch of the tree will result in a tree which is incomplete in sensor **b** (circled).

above, there are $k-2$ nodes that need to be checked for completeness. If a subtree (in τ_{sub1}) at depth r, denoted by τ_{r1} and containing n_{1k} becomes identical to its sibling subtree τ_{r2} after the removal of n_{1k}, then we can not merge n_{1k}. Instead, if n_{1k} has a sibling node, n_{2k} (obviously it would also be at depth k), we try removing its exact counterpart, n'_{2k} in τ_{r2}. Again in this case, we need to check at most $k-2$ nodes in the tree for completeness, but we know for sure that at least τ_{r1} cannot be identical to τ_{r2} after the removal of n'_{2k} because of the presence of n_{1k} in τ_{r1}. Therefore, there are just $k-3$ nodes that we need to check for the completeness constraint for the proposed removal of n'_{2k}. If the sibling of n_{1k} is a leaf and its parent node is m_{1k}, then we try removing its counterpart m'_{2k} in τ_{r2}. In this case, again we need to check $k-3$ nodes for completeness constraint. By continuing in this fashion for t such steps, suppose we reach a node n_{1p} $(k-t \leq p \leq k)$. At this point there will be at most $k-t-2$ nodes to check for completeness for the proposed removal of n_{1p}. Suppose that the removal of n_{1p} results in the tree becoming incomplete at certain higher node at depth s. Then according to our algorithm, if n_{2p} is the sibling node of n_{1p}, we will try to merge its counterpart node n'_{2p} (with $(k-t-2)-1$ completeness constraints) in σ_2. But, if both the children of n'_{2p} are non-leaf nodes, we do not merge n'_{2p}. In that case we try to merge the deepest node in the subtree whose root node is n'_{2p}. If the depth of that node is q $(p \leq q \leq k)$, then we have at most $(k-t-2)-1+(q-p)$ completeness constraints for the removal of that node. Therefore, even in the worst case, when $p = k-t$ and $q = k$, $(k-t-2)-1+(q-p) = (k-t-2)-1+(k-(k-t)) = k-3$. Therefore, by continuing in this fashion, we can reach, by induction, to a node which requires

$(k - k) = 0$ nodes to be checked for completeness, and hence can be merged without making the tree incomplete in any node. After that, we repeat this procedure again to one of the deepest node of the subtree that we want to merge, until the entire subtree is merged. Next, let us consider the special case when $r = 1$ (that is $\tau_{r2} = \tau_{sub2}$). An interesting point to note here is that during the entire procedure of merging τ_{sub1}, this situation can occur at most once. This is so because τ_{sub2} is retained completely during the entire procedure while nodes from τ_{sub1} are continuously removed. Therefore, τ_{sub1} can become exactly identical to τ_{sub2} only when the number of nodes in τ_{sub1} is equal to that in τ_{sub2}. However, the way we merge a subtree takes care that such a situation never happens. For example, we always merge a subtree that has fewer nodes in it than the other. This will become clearer from discussion later in this section. The following pseudocode summarizes the *smartMerge* algorithm.

Algorithm 4 *smartMerge*

```
1   Initialize n_1k ← maximum depth node in τ_sub1
2   while τ_sub1 is not a leaf node, do
3       flag_delete ← TRUE
4       for r = 1 to k − 2 in the path containing n_1k, do
5           if τ_r1 = τ_r2, then
6               if n'_2k (the counterpart node of the sibling node of n_1k), exists, then
7                   k ← q (maximum depth in τ_n'2k)
8                   n_1k ← n'_2q
9               else
10                  k ← k − 1
11                  n_1k ← m'_2k (the counterpart node of the parent node of n_1k)
12              end if
13              flag_delete ← FALSE
14              break
15          end if
16      end for
17      if flag_delete = TRUE
18          merge n_1k
19          n_1k ← maximum depth node in τ_sub1
20      end if
21  end if
```

For reducing any given arbitrary tree to some simple tree, we start at the root node, merge one of its left or right subtrees (using *smartMerge*) and then repeat the same procedure on the next deeper node and so on until we reach a leaf node. By doing this

iteratively, we can guarantee to arrive at a simple tree σ_1. Let s_k be the current sensor whose subtree we want to merge where k is the depth of s_k, such that $1 \leq k \leq n$. $k = n$ means that both the children of s_k are leaves and hence we do not need to merge. For $1 \leq k < n$ one of following cases must be true:

Case 1: Both left and right subtrees are complete in $n - k$ sensors: In this case, the removal of any of the subtrees will not affect the completeness of the overall tree. Therefore we use the *smartMerge* algorithm to merge the subtree which has fewer nodes in it. If both the subtrees have equal number of nodes, then we can merge any one of the two.

Case 2: One of the left and right subtrees is complete in $n-k$ sensors: In this case, since one of the subtrees is complete in $n-k$ sensors, we can be sure that the removal of the other subtree will not affect the overall completeness of the tree. Therefore we use the *smartMerge* algrithm to merge the subtree which is not complete in $n - k$ sensors.

Case 3: None of the left and right subtrees is complete in $n-k$ sensors: In this case, we cannot merge any one of the subtrees until we make sure that the other subtree is made complete in $n - k$ sensors. Therefore, we select to retain the subtree which has a larger number of different sensors in it. If both the subtrees have an equal number of different sensors, then we choose to merge the subtree that has fewer nodes. If the two subtrees have equal number of different sensors and equal number of nodes, we can merge any one of those two subtrees. Before doing that, we iteratively insert the remaining sensors at one of the deepest nodes of the subtree that we decide to retain, till that subtree is complete in $n - k$ sensors. Then we use the *smartMerge* algorithm to merge the other subtree. Figure 4.6 (trees numbered **(1)** through **(23)**) shows an example of obtaining a simple tree from an arbitrary tree in τ^6 by repeated use of the *smartMerge* algorithm.

Finally, we prove that any simple tree σ_2 in τ^n can be reached from any other simple tree σ_1 in τ^n using the four operations repeatedly. Let P_1 and P_2 be the essential paths

of simple trees σ_1 and σ_2 respectively, where:

$$P_1 = s_1 \xrightarrow{d_1} s_2 \xrightarrow{d_2} \dots \xrightarrow{d_{n-1}} s_n$$
$$P_2 = s'_1 \xrightarrow{d'_1} s'_2 \xrightarrow{d'_2} \dots \xrightarrow{d'_{n-1}} s'_n$$

where $s_1, s_2, \dots, s_n$ are sensors at depth $1, 2, \dots, n$ in the essential path of σ_1 and $s'_1, s'_2, \dots, s'_n$ are sensors at depth $1, 2, \dots, n$ in the essential path of σ_2. Also, $D_1 = \{d_1, d_2, \dots, d_{n-1}\}$ and $D_2 = \{d'_1, d'_2, \dots, d'_{n-1}\}$ are direction $(n-1)$-tuples such that $d_i, d'_i \in \{Left, Right\}, i = 1, \dots, n$. Also, we use $\overline{d}_i$ to denote the direction complementary to d_i, that is, $\overline{d}_i = Left \Leftrightarrow d_i = Right$ and vice-versa. Lastly we say that $D_1 = D_2 \Leftrightarrow d_i = d'_i, i = 1, \dots, n$.

In order to go from σ_1 to σ_2, we first modify σ_1 such that $D_1 = D_2$ Let k be an integer such that $1 \leq k \leq n$ and

$$d_i = \begin{cases} d'_i & \text{if } 1 \leq i < k \\ \overline{d'}_i & \text{if } i = k \text{ and } k < n \end{cases}$$

Note that if $k = n$ (i.e. $D_1 = D_2$), then σ_1 can be transformed to σ_2 only by repeated use of swap operation. If $k = n - 1$, then D_1 differs from D_2 only in d_{n-1}. In this case we temporarily add s_n towards $\overline{d}_1$ at s_1, then merge s_n from P_1, re-insert s_n towards $\overline{d}_{n-1}$ at s_{n-1} and finally merge s_n from $\overline{d}_1$ at s_1. Lastly, if $k < n-1$, we add s_n towards d'_k at s_k (because $\overline{d}_k = d'_k$) and merge s_n from P_1. We then add s_{n-1} at s_n towards d'_{k+1} and merge s_{n-1} from P_1. We repeat this procedure for all $k \leq i < n-1$ until $D_1 = D_2$. After that we rearrange the sensors in σ_1 using repeated swap operations so that σ_1 becomes exactly identical to σ_2. Figure 4.5 and Figure 4.6 (trees numbered **(23)** through **(42)**) show an example of the method described above. In this way, we prove that any simple tree can be reached from any other simple tree, using neighborhood operations repeatedly in τ^n.

Figure 4.6 (trees numbered **(42)** through **(55)**) also shows an example of reaching to an arbitrary tree from a simple tree. Notice that all the steps in this sequence are

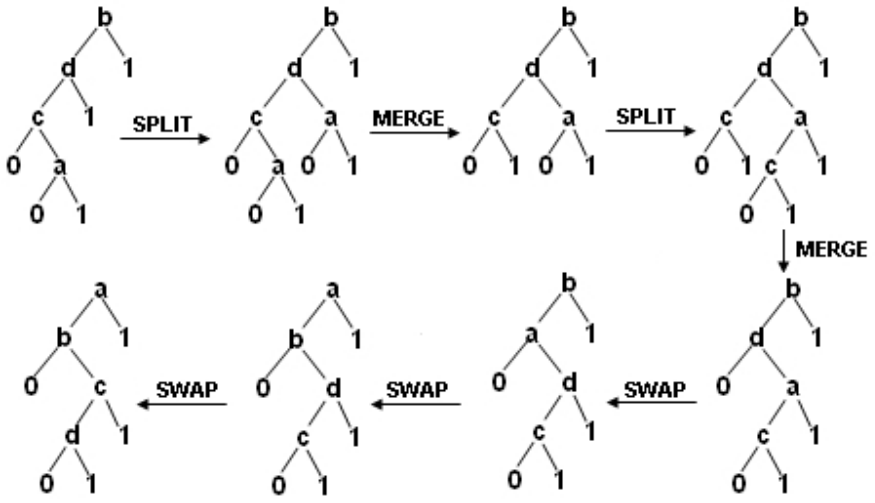


Figure 4.5: An example showing that any simple tree can be reached from any other simple tree using the four neighborhood operations repetitively. The first tree in the chain is σ_1 and the last one is σ_2.

reversible, hence showing that the entire *smartMerge* algorithm is exactly reversible. This completes the proof.

4.2 Tree Space Traversal

We have explored alternate ways to search for a minimum cost trees in the entire CM tree space. In this section we describe two such methods. Please see [24] and [25] for more details about these algorithms.

4.2.1 The Stochastic Search Method

Our initial approach to search for the minimum cost trees using the the above mentioned neighborhood operations was a simple greedy search - randomly start at any arbitrary tree in the space, find its neighboring trees using the above operations, move to the neighbor with the lowest cost, and then iterate. As expected, however, the cost function is multimodal and the greedy strategy gets stuck at local minima. For example, there are 9 modes in the entire space of 114 trees for 3 sensors and 193 modes in the space of 66,600 trees for 4 sensors. To address the problem of getting stuck in a local

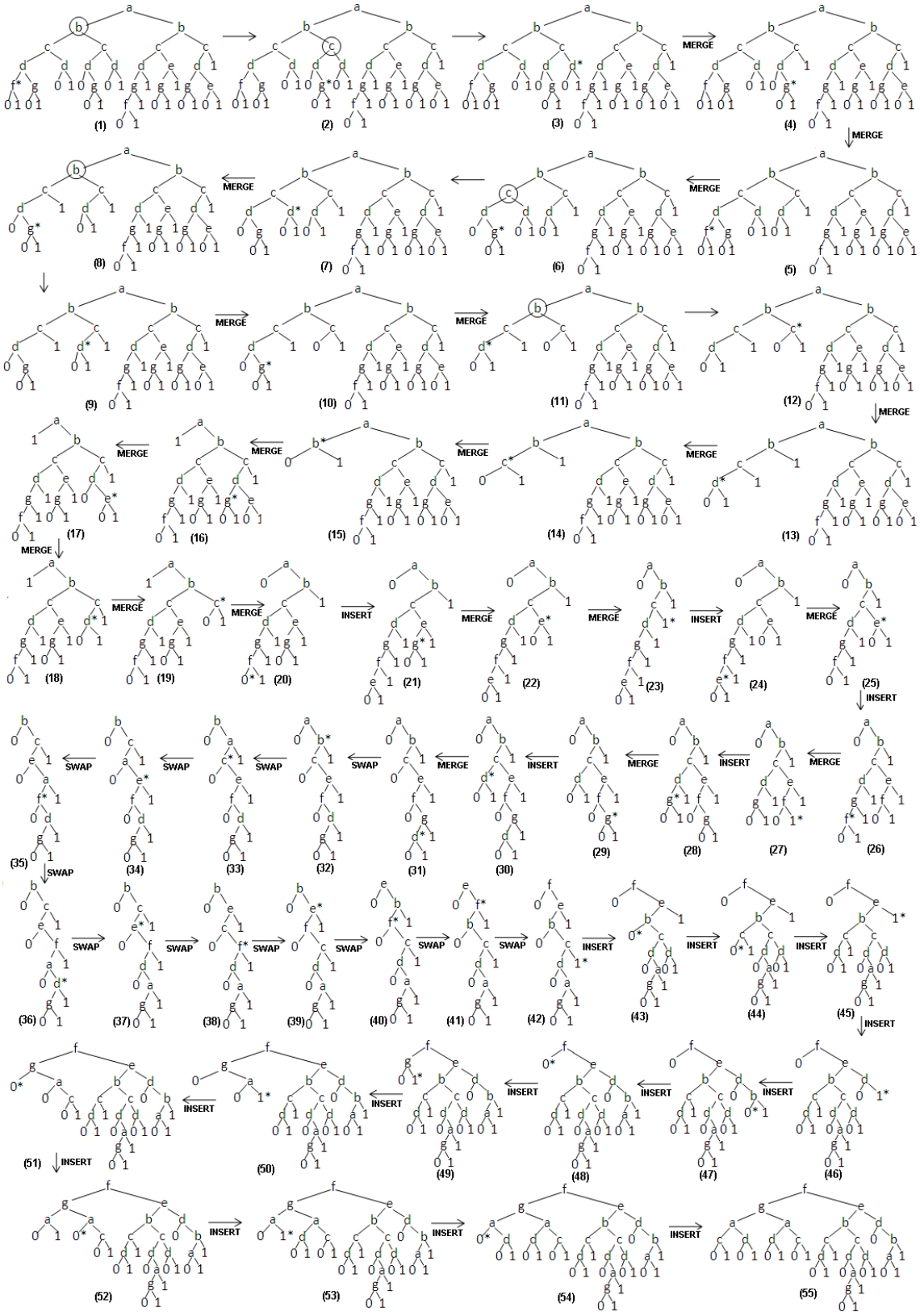


Figure 4.6: An example showing that any arbitrary tree in τ^6 can be reached from any other arbitrary tree using the four neighborhood operations repeatedly. The sensor marked * in every tree is subject to a neighborhood operation while the circled sensors show a possible conflict with completeness constraint.

minimum, we developed a stochastic search algorithm coupled with simulated annealing. The algorithm is stochastic insofar as it selects moves according to a probability distribution over neighboring trees. The simulated annealing aspect involves a so-called "temperature" t, initiated to one and lowered in discrete unequal steps after every m hops until we reach a minimum. Specifically, if we are at the ith tree τ_i, then the probability of going to its kth neighbor, denoted τ_{ik}, is given by the following equation.

$$P_{ki} = \frac{(f(\tau_i)/f(\tau_{ik}))^{1/t}}{\sum_{j=1}^{n_i} (f(\tau_i)/f(\tau)_{ij})^{1/t}} \tag{4.1}$$

where $f(\tau_i)$ and $f(\tau_{ij})$ are the costs of trees τ_i and τ_{ij}, respectively and n_i is the number of trees in the neighborhood of τ_i. Therefore, as the temperature is decreased, the probability of moving to the least expensive tree in the neighborhood increases. The following algorithm describes the stochastic search method for finding minimum cost trees.

Algorithm 5 Stochastic Search Method using Simulated Annealing

```
1   for p = 1 to numberOfStartPoints do
2       t ← 1
3       numberOfHops ← 0
4       currentTree ← random(allTrees)
5       do
6           Compute c_i
7           neighborTrees ← findNeighborTrees(currentTree)
8           for all 1 ≤ k ≤ n_i do
9               Compute c_ik
10              Compute P_ki
11          end for
12          currentTree ← random(neighborTrees, P_ki)
13          numberOfHops ← numberOfHops +1
14          if numberOfHops = m then
15                  t ← t − Δt
16              numberOfHops ← 0
17          end if
18      while c_i > c_ik ∀ 1 ≤ k ≤ n_i
19  end for
20  Output lowest cost tree over all p
```

4.2.2 Genetic Algorithms based Search Method

We have also used a genetic algorithm (GA) based approach to search CM tree space. The underlying concept of this approach is to obtain a population of "better" trees from an existing population of "good" trees by performing three basic genetic operations on them - Selection, Crossover, and Mutation. In our application, "better" decision trees correspond to lower cost decision trees than the ones in the current population. As we keep on generating newer generations of "better" trees (or currently best trees), the gene pool, *genePool*, keeps on increasing in size. We describe each of the genetic operations in detail below. The use of GAs to explore tree spaces was also considered by Papagelis and Kalles [28] and Fu [17].

1. *Selection*: We select an initial population of trees, *bestPop*, randomly out of the CM tree space to form a gene pool. We always maintain a population of size N of the lowest cost trees out of the whole population for the crossover and mutation operations.

2. *Crossover*: The crossover operations are performed between every pair of trees in *bestPop*. For each crossover operation between two trees τ_i and τ_j, we randomly select nodes s_1 and s'_1 in τ_i and τ_j respectively and replace the subtree τ_{is_1} (rooted at s_1 in τ_i) with $\tau_{js'_1}$ (rooted at s'_1 in τ_j). A typical crossover operation is shown using the example in Figure 4.7.

   We randomly perform these crossovers repeatedly, resulting in k distinct trees (or *all* distinct trees k' if $k' < k$) from a pair of trees. However, we impose some restrictions on the random selection of the nodes to make sure that the resultant tree obtained after the crossover operation also lies in the CM tree space. For example, if τ_{is_1} is a right subtree, then $\tau_{js'_1}$ cannot be a 0 leaf. Similarly, if τ_{is_1} is a left subtree, then $\tau_{js'_1}$ cannot be a 1 leaf. These restrictions ensure that the resulting tree would also be a monotonic tree. To make sure that the resulting tree is complete, we impose two restrictions: the sibling subtree of τ_{is_1}, which is denoted by τ_{is_2}, should not be exactly identical to $\tau_{js'_1}$ and $\tau_{js'_1}$ should have all the sensors which the tree τ_i would lack, once τ_{is_1} is removed from it. In other

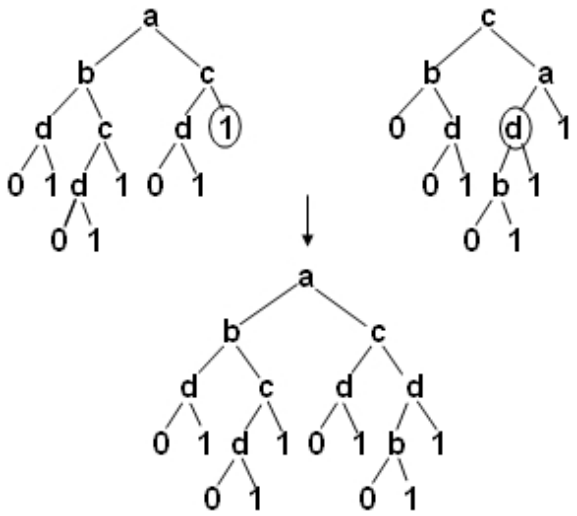


Figure 4.7: An example of the crossover operation.

words, the tree resulting from the crossover operation should have all the sensors present in it.

3. *Mutation*: The mutation operations are performed after every m generations of the algorithm. We do two types of mutations. The first type of mutation operation consists of generating all the neighboring trees of the current best population of trees using the four neighborhood operations described earlier and putting them into the gene pool. The second type of mutation operations consists of replacing $\left(\frac{1}{M}\right)$th of the trees in *bestPop* with random samples from the CM tree space which are not in the gene pool, therefore increasing the probability of generating trees that are quite different from the ones in the current gene pool.

The following algorithm explains the GA based search method for finding the minimum cost trees.

Algorithm 6 Genetic Algorithms based Search Method

```
1    Initialize bestPop ← generateTreesRandomly(N)
2    Initialize genePool ← bestPop
3    Initialize lastMutation ← 0
4    for p = 1 to totalNumberOfGenerations do
5        for all τ_i, τ_j ∈ bestPop, i ≠ j
6            GATrees ← generateGATreesRandomly(τ_i, τ_j, k)
7            genePool ← genePool ⋃ GATrees
8        end for
```

```
9       bestPop ← selectBestTrees(genePool, N)
10      lastMutation ←lastMutation + 1
11      if lastMutation = m then
12          for all τ ∈ bestPop do
13              neighborTrees ← findNeighborTrees(τ)
14              genePool ←genePool ⋃ neighborTrees
15          end for
16          bestPop ← selectBestTrees(genePool, N)
17          bestPop ← selectBestTrees(bestPop, N − N/M)
18          bestPop ←bestPop ⋃ generateTreesRandomly(N/M)
19          genePool ←genePool ⋃ bestPop
20          lastMutation ← 0
21      end if
22  end for
23  Ouput bestPop
```

Chapter 5

Shrinking the Tree Space

As we have commented earlier, the number of trees increase more than double exponentially with the number of sensors types. In Chapter 3 we looked into the notions of monotonicity and completeness. We noticed that these notions are theoretical assumptions to lower the number of potentially cheap trees by an appreciable factor. Further, in Chapter 4 we molded the problem for finding the optimum tree as a search problem through a space of trees which is irreducible under the defined neighborhood operations. Also, we discussed GA based algorithms as another experimental technique to enable us to search for the optimum tree(s) by evaluating as fewer number of trees as possible.

In this chapter we look at yet another theoretical aspect of the trees. We introduce the ideas of "equivalence" of two or more trees and the "irreducibility" of a BDT and try to show how these ideas help us reduce the tree space into the space of classes of equivalent trees. We also do a more rigorous theoretical analysis of the definitions of completeness and monotonicity. Later, we modify and apply the search algorithms discussed in Chapter 4 on the space of equivalence classes of trees. The experimental results of this chapter together with those from the previous chapters are combined together and presented in Chapter 6.

5.1 Tree Equivalence

Equivalence of decision trees is an interesting property and has been discussed quite extensively in [11], [22] and [32]. In general, equivalence of two or more decision trees corresponds to their being "decision-equivalent". We, on the other hand, are interested in a definition of equivalence that results in smaller equivalence classes. We obtain this subclass by putting an additional cost constraint on definition of decision equivalent

trees and hence define the "cost-equivalence" of two or more decision trees. In the following sub-sections, we describe both the notions of equivalence. Later, we also define a canonical representation for the class of cost-equivalent decision trees.

5.1.1 Decision Equivalent BDTs

Most of the machine learning literature that talks about equivalence classes of decision trees, focuses its interest on two or more trees being decision-equivalent. For example, [32] defines the concept of equivalence for trees as follows: two decision trees are equivalent if and only if they represent the same hypothesis. Also, [10] and [11] describe different ways, in which two or more decision trees can be equivalent. Finally, given two decision trees, [32] also discusses a fast algorithm to establish if they are decision equivalent or not. In other words, two or more decision trees are called decision-equivalent if their underlying Boolean function is same. That is, two or more decision trees are called decision-equivalent if and only if they give exactly same outputs for similar sets of input attribute values. Therefore, under such a setting, it does not matter how the attributes are arranged relative to each other in a tree, all that matters is the final output of the tree. However, in our case, since the attributes are sensors, we need to put an additional constraint that the threshold value for any sensor should be the same across all the trees. Figure 5.1 shows an example of the trees that are decision-equivalent to one another and their Boolean function $F(\mathbf{a}, \mathbf{b}, \mathbf{c}) = 00011111$ represents the equivalence class that they belong to. (Please note that we can form many more different trees with the same boolean function by relaxing the completeness constraint.) Also note that all these trees are equally "efficient" in their classification (i.e., their misclassification rates are equal) as long as all the sensors have fixed threshold values across all the trees.

5.1.2 Cost Equivalent BDTs

Decision-equivalent trees, in general, can have different costs because of the difference of number of sensors and their relative positions across different trees. To extend the definition of decision-equivalence to cost-equivalence, we put an additional cost

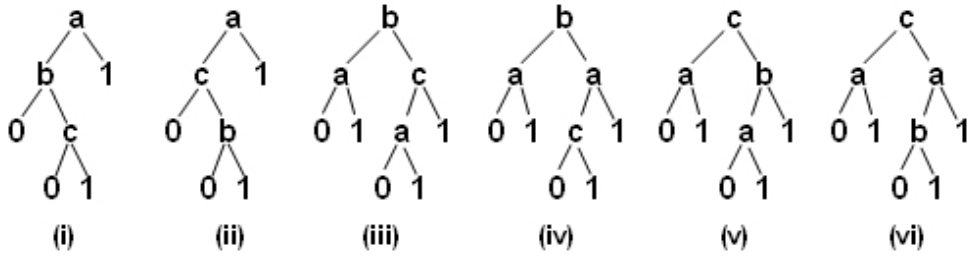


Figure 5.1: An example of decision-equivalent binary decision trees.

constraint on the trees. Therefore, saying loosely, two or more trees are called cost-equivalent if and only if they are decision-equivalent and have exactly same costs for same values of sensor threshold values. This additional constraint gives rise to a very interesting relationship between the structures of all the trees belonging to the same cost-equivalence class. To understand this structural relationship between the trees, we borrow the definition of *Transposition* from [11]. But before that, we briefly explain the definitions of a "path" (or "spine") in a tree and "bag-equivalence" of two paths or two sets of paths first. We will be using these notions throughout the rest of this chapter.

Path: A path, P in a tree is defined as a sequence of *(attribute, output)* pairs (starting from the root node till the last sensor above a leaf node in that branch), together with the label of the leaf node. Therefore,

$$P = \left[\left((s_1, Y_{s1}), (s_2, Y_{s2}), ..., (s_k, Y_{sk})\right), l\right] \tag{5.1}$$

where, s_i is the ith sensor (attribute) from root, Y_{si} is the output of the that sensor, k is the total number of sensors in the path and l is the label of the leaf node. In the case of attributes with discrete outputs, both Y_{si} and l belong to a finite set of possible output values. For example, in our case of binary decision trees, $Y_{si} \in \{left, right\}$ and $l \in \{0, 1\}$. Also, the set of all the paths in a tree is called the path-set and is denoted by $\mathbf{P}$.

Bag-equivalence: Two paths P^1 and P^2, both of length k, are called bag-equivalent if

and only if the sets,

$$\{(s_1^1, Y_{s1}^1), (s_2^1, Y_{s2}^1), ..., (s_k^1, Y_{sk}^1)\} = \{(s_1^2, Y_{s1}^2), (s_2^2, Y_{s2}^2), ..., (s_k^2, Y_{sk}^2)\}, \text{ and}$$

$$l^1 = l^2.$$

Now, trivially, two path-sets are called bag-equivalent if the above two identities hold for *all* the paths in the two sets. Note, however, that two bag equivalent paths may not be identical because of the different order of sensors in the two paths. Therefore, we can now define transposition property of two or more trees quite easily as follows.

Transposition: Two (or more) trees are called transposition-equivalent or simply, *transposes* of each other, if and only if their path-sets are bag-equivalent.

This gives us an alternate and more precise definition of cost-equivalent decision trees - two or more trees are called cost-equivalent if and only if[1] they are transposition equivalent to one another. (Therefore, we use the terms "cost-equivalence" and "transposition-equivalence" interchangeably). The proof of this fact is quite intuitive because the transposition-equivalent trees have bag-equivalent path-sets. Therefore, their cost functions must be exactly identical to one another and so do their optimum costs. For example, Figure 5.2 shows an example of two cost-equivalent trees. Notice that their path-sets are bag-equivalent.

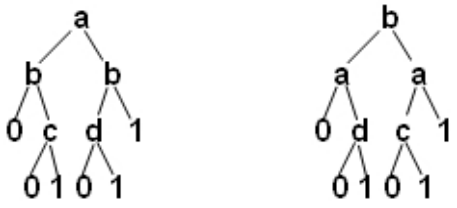


Figure 5.2: An example of cost-equivalent binary decision trees.

The size of a class of cost-equivalent trees varies for different trees in the tree space, but the size of the largest equivalence class also grows double-exponentially with the number of sensor types. Since one of the trees belonging to its equivalence class is

[1]While we state this, we ignore the case where two non-transpose trees might have identical cost spuriously.

sufficient to represent the entire class, we can now have a tree space which consists of classes of cost-equivalent trees, instead of individual trees themselves. This gives us a significant potential reduction in the size of the tree space we want to search. However, we still do not have a unique representation of each of these equivalence classes. Before we establish a unique, canonical representation of a class of cost-equivalent trees, we need to look at some other issues, like how a decision tree can be reduced to its minimal form and how transpose-equivalence disturbs the definitions of completeness and monotonicity. We shall do that in the following sections.

5.2 Revisiting Monotonicity

In this section, we will see how the definition of completeness and monotonicity are affected by the concept of transposition. Since the tree space now comprises equivalence classes of cost-equivalent trees, we now want the definitions of completeness and monotonicity to suit these equivalence classes rather than the trees themselves. For example, a tree that was considered complete and monotonic earlier, might have a cost-equivalent tree which is incomplete or non-monotonic or both. Therefore we redefine a monotonic decision tree as follows. To avoid confusion with the previous definition of monotonicity, we call this new class of monotonic decision trees as "equi-monotonic" decision trees.

Equi-monotonic Decision Trees. A binary decision tree will be called *equi-monotonic* if, in all the trees belonging to its class of cost-equivalent trees, all leaf nodes emanating from a left branch are labeled 0 and all leaf nodes emanating from a right branch are labeled 1.

Figure 5.3 shows an example of an equivalence class of trees. Trees (i) and (ii) were previously considered monotonic, are now considered non-monotonic because they have an equivalent tree (iii), which is clearly non-monotonic in **a**.

We now take a closer look at the definition of monotonicity. If we lift the constraint that every instance of the same sensor in a tree should have identical thresholds, we can easily prove that a non-monotonic tree in that scenario, can never be an optimum tree. But before going to the actual proof, we would like to define a "general non-monotonic

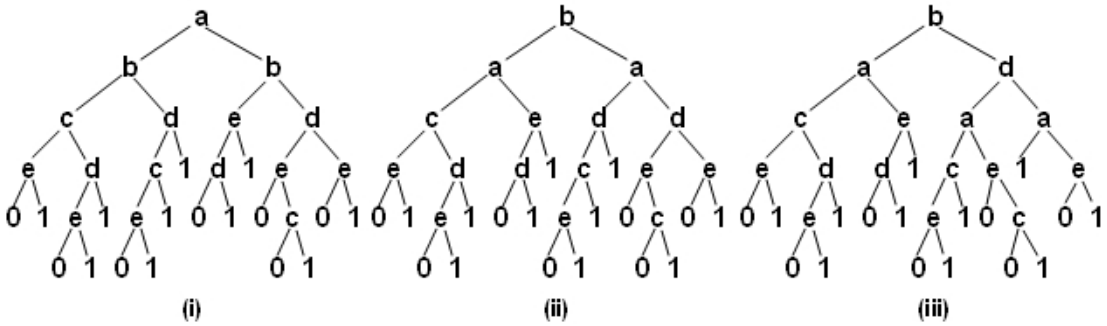


Figure 5.3: An example of equivalence class of trees non-monotonic in sensor **a**.

tree" formally as follows.

General Non-monotonic Binary Decision Trees. A binary decision tree will be called a general non-monotonic BDT if different instances of a sensor in that tree can be set to different thresholds and the tree has at least one sensor whose left leaf node is labeled 1 or whose right leaf node is labeled 0 or both.

Therefore, in the most general case where every instance of a sensor in a tree can have different threshold.

Theorem 5.1. *For a finite values of sensor thresholds, a general non-monotonic binary decision tree, can never be the cheapest binary decision tree.*

Proof: We will start by proving the theorem for a single sensor tree. Consider a monotonic tree τ' with sensor s shown as the left tree in Figure 5.4. Let T'_s be its current (or even optimum) threshold value. The various detection probabilities of τ' are shown in different colors[2]. Also consider a monotonic counterpart τ shown to the right. Further, let us assume that we set the value of its threshold, T_s such that $P_{s=0|0} = P'_{s=1|0}$ (and $P_{s=1|0} = P'_{s=0|0}$). Therefore, in Figure 5.4, the area of the yellow region in (i) is equal to that of the purple region[2] in (ii) and similarly the area of the purple region in (i) is equal to that of the yellow region in (ii). Also notice that there is a tiny blue region to the left of the threshold in (ii).

[2]A part of the green region overlaps with the purple one. Basically, the entire region inside the second Gaussian on the right side of the threshold should be considered green. Similarly, the entire region inside the first Gaussian to the left of the threshold should be considered yellow.

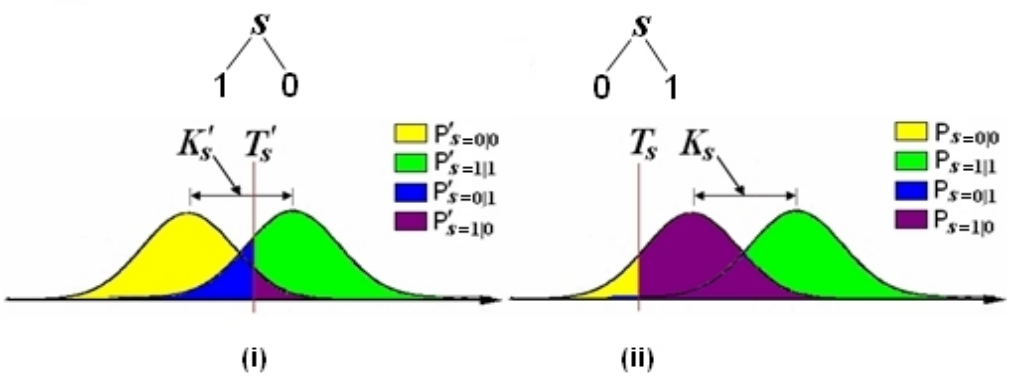


Figure 5.4: An example of a one sensor non-monotonic tree and its corresponding monotonic tree.

For any finite value of the threshold T_s' selected, $P_{s=0|1} < P'_{s=1|1}$ if we always use the above mentioned condition to choose the value of T_s. Therefore, the respective cost functions of τ' and τ can be written as follows.

$$f(\tau') = P_0 C_{\mathbf{s}} + P_1 C_{\mathbf{s}} + P_0 P'_{\mathbf{s}=0|0} C_{FP} + P_1 P'_{\mathbf{s}=1|1} C_{FN} \tag{5.2}$$

$$f(\tau) = P_0 C_{\mathbf{s}} + P_1 C_{\mathbf{s}} + P_0 P_{\mathbf{s}=1|0} C_{FP} + P_1 P_{\mathbf{s}=0|1} C_{FN}. \tag{5.3}$$

We claim that $f(\tau) < f(\tau')$, therefore, from equations 5.2 and 5.3,

$$P_0 P_{\mathbf{s}=1|0} C_{FP} + P_1 P_{\mathbf{s}=0|1} C_{FN} < P_0 P'_{\mathbf{s}=0|0} C_{FP} + P_1 P'_{\mathbf{s}=1|1} C_{FN}.$$

Now, since $P'_{s=1|0} = P_{s=0|0}$, therefore the above equation, without loss of generality, becomes

$$P_1 P_{\mathbf{s}=0|1} C_{FN} < P_1 P'_{\mathbf{s}=1|1} C_{FN}$$

or,

$$P_{\mathbf{s}=0|1} < P'_{\mathbf{s}=1|1}$$

which, as discussed above, is true. Also, since $T_s' = \infty \Rightarrow T_s = -\infty$, therefore,

$$P_{\mathbf{s}=0|1} = P'_{\mathbf{s}=1|1} = 0$$

and hence,

$$f(\tau) = f(\tau').$$

That means that we assign every input a label 1, which obviously becomes independent of $\mathbf{s}$ in both τ' and τ. Similarly, $T'_s = -\infty, \Rightarrow T_s = \infty$, and again,

$$f(\tau) = f(\tau')$$

Since we assign every input a label 0 it is again independent of τ' or τ. Other possibilities of a non-monotonic tree τ' are shown as follows.

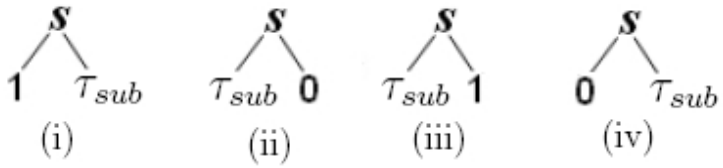


Figure 5.5: Other possibilities of non-monotonic trees (i) and (ii), and their corresponding monotonic trees (iii) and (iv) respectively.

As shown in Figure 5.5 (i) and (ii) a tree can be non-monotonic in a sensor s in two ways (i) sensor s has a 1 to its left and a subtree to its right, or (ii) sensor s has a 0 to its right and a subtree to its left. In case (i) we can get the monotonic counterpart tree, τ (Figure 5.5(iii)) of τ' by switching the right subtree of s with the 1 on the left. In this tree we can again set the value of T_s such that $P'_{s=0|0} = P_{s=1|0}$ and leave the threshold values of all other sensors unchanged. Using a similar analysis as above, we can show that without any loss of generality, $f(\tau) < f(\tau')$. In case (ii) we can get the monotonic counterpart tree, τ (Figure 5.5(iv)) of τ' by switching the left subtree of s with the 0 on the right. In this tree, we can set the value of T_s such that $P'_{s=1|1} = P_{s=0|1}$ and leave all other sensor thresholds unchanged. Again, writing down the cost functions of the trees τ' and τ, we can easily prove that $f(\tau) < f(\tau')$. This completes the proof.

Note, however, in practice, we put a constraint on the sensors of same type to have identical thresholds. In that case the above theorem doesn't necessarily hold. For example, we can have multiple instances of a sensor in a tree, one or more (but not all) of which can be non-monotonic. In this case, we cannot guarantee to have a tree

cheaper than that non-monotonic tree. Figure 5.6 (i) shows an example of one such tree.

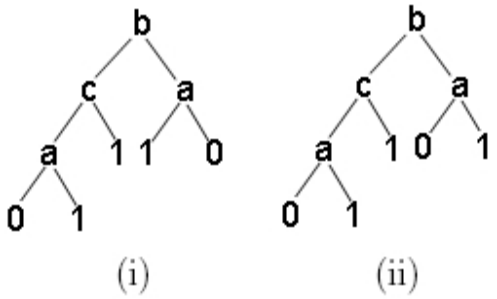


Figure 5.6: An example of a non-monotonic tree τ' that can be an optimum tree and its corresponding monotonic tree τ. Only one of the two instances of sensor **a** is monotonic in τ'.

Next we prove that the tree shown in Figure 5.6 (i) can infact be the cheapest cost tree. We compare this tree τ' with its corresponding monotonic tree τ (Figure 5.6 (ii)). We know that in general, τ can be the cheapest cost tree. Therefore, if we can prove that τ' can be cheaper than τ even for one specific choice of sensor thresholds in τ, that should be a sufficient argument to prove that trees like τ' are potentially valid candidates for the cheapest trees and thus should be included in the search space. To start, we compare the cost functions of the two trees. The individual cost functions of the two trees are given by the following equations. Here, $f(\tau)$ is the cost function for τ while $f(\tau')$ is the cost function of τ' with all sensors set to the same threshold values as τ.

$$\begin{aligned} f(\tau') \;=\; & P_0\left(C_{\mathbf{b}} + P_{\mathbf{b}0|0}C_{\mathbf{c}} + P_{\mathbf{b}0|0}P_{\mathbf{c}0|0}C_{\mathbf{a}} + P_{\mathbf{b}1|0}C_{\mathbf{a}}\right) \\ & + P_1\left(C_{\mathbf{b}} + P_{\mathbf{b}0|1}C_{\mathbf{c}} + P_{\mathbf{b}0|1}P_{\mathbf{c}0|1}C_{\mathbf{a}} + P_{\mathbf{b}1|1}C_{\mathbf{a}}\right) \\ & + P_0\left(P_{\mathbf{b}0|0}P_{\mathbf{c}1|0} + P_{\mathbf{b}0|0}P_{\mathbf{c}0|0}P_{\mathbf{a}1|0} + P_{\mathbf{b}1|0}P_{\mathbf{a}0|0}\right)C_{FP} \\ & + P_1\left(P_{\mathbf{b}0|1}P_{\mathbf{c}0|1}P_{\mathbf{a}0|1} + P_{\mathbf{b}1|1}P_{\mathbf{a}1|1}\right)C_{FN} \end{aligned} \tag{5.4}$$

$$
\begin{aligned}
f(\tau) &= P_0\left(C_{\mathbf{b}} + P_{\mathbf{b}0|0}C_{\mathbf{c}} + P_{\mathbf{b}0|0}P_{\mathbf{c}0|0}C_{\mathbf{a}} + P_{\mathbf{b}1|0}C_{\mathbf{a}}\right) \\
&\quad + P_1\left(C_{\mathbf{b}} + P_{\mathbf{b}0|1}C_{\mathbf{c}} + P_{\mathbf{b}0|1}P_{\mathbf{c}0|1}C_{\mathbf{a}} + P_{\mathbf{b}1|1}C_{\mathbf{a}}\right) \\
&\quad + P_0\left(P_{\mathbf{b}0|0}P_{\mathbf{c}1|0} + P_{\mathbf{b}0|0}P_{\mathbf{c}0|0}P_{\mathbf{a}1|0} + P_{\mathbf{b}1|0}P_{\mathbf{a}1|0}\right)C_{FP} \\
&\quad + P_1\left(P_{\mathbf{b}0|1}P_{\mathbf{c}0|1}P_{\mathbf{a}0|1} + P_{\mathbf{b}1|1}P_{\mathbf{a}0|1}\right)C_{FN}. \qquad (5.5)
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
f(\tau') - f(\tau) &= P_0\left(P_{\mathbf{b}1|0}\left(P_{\mathbf{a}0|0} - P_{\mathbf{a}1|0}\right)\right)C_{FP} \\
&\quad + P_1\left(P_{\mathbf{b}1|1}\left(P_{\mathbf{a}1|1} - P_{\mathbf{a}0|1}\right)\right)C_{FN}. \qquad (5.6)
\end{aligned}
$$

Let us assume that the optimum threshold value of sensor $\mathbf{a}$, $T_{\mathbf{a}} > \mu_2$ (as shown in Figure 5.7), where μ_2 is the mean of the second Gaussian. This, in practice, is quite possible if $P_0 >> P_1$ and the distribution of samples as seen by sensor $\mathbf{a}$ (in the left subtree of sensor $\mathbf{b}$) consists of almost all the negative samples and almost none of the positive samples.

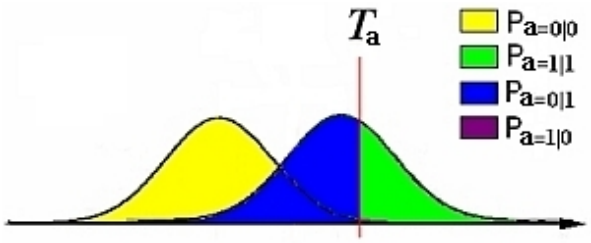


Figure 5.7: An example of possible optimum threshold value of sensor $\mathbf{a}$.

Clearly from Figure 5.7, $P_{\mathbf{a}0|0} > P_{\mathbf{a}1|0}$ and $P_{\mathbf{a}0|1} > P_{\mathbf{a}1|1}$. If $P_1 P_{\mathbf{b}1|1} C_{FN} >> P_0 P_{\mathbf{b}1|0} C_{FP}$, then it is possible that $f(\tau') < f(\tau)$. This completes the proof for the example.

5.3 Revisiting Completeness

Revising the definition of completeness to account for cost equivalence presents two specific problems. Firstly, as we noticed, the definition of completeness in Chapter 2, Section 3.2.1 only accounts for the left and right subtrees of any node being *exactly*

identical to each other. But what if they are cost-equivalent to one another? In that case either of the subtrees can be rearranged using just the transposition operation(s) to look exactly identical to the other subtree, resulting in the overall tree becoming incomplete at that particular node. Therefore, a tree should not have any non-leaf node in it with cost-equivalent subtrees. Secondly, in a complete tree, we allowed the nodes to have subtrees with same Boolean function, (as long as the subtrees are not exactly identical). For example, in Figure 3.1, we defined trees (i) and (ii) as complete trees because they do not have any nodes with similar left and right subtrees. However, in both the trees the sensor **a** has subtrees with same Boolean function, $F(\mathbf{b}, \mathbf{c}) = 0111$. Therefore, according to the definition of decision-equivalent trees, the two subtrees are decision equivalent and hence are equally efficient in their classification of containers into good and bad. But, in general, they can have different cost. Therefore, we can always replace the higher cost subtree with the lower cost one, to achieve a tree with lower overall cost, without affecting the overall efficiency of the tree. In that case, the resultant tree would become exactly identical to either tree (iii) or (iv) in Figure 3.1 and thus would be incomplete in sensor **a**. These considerations lead to the following definitions.

Equi-complete Decision Trees. A binary decision tree will be called *equi-complete* if every sensor occurs at least once in the tree and, at any non-leaf node in the tree, the left and right subtrees do not correspond to same Boolean function.

Following this definition, it is quite trivial to prove that all equi-complete trees correspond to complete Boolean functions (Chapter 2, Section 3.1.1). However, in practice, it is not trivial to verify if a given tree according to this newer definition of completeness, is complete or not. Enumerating all equivalent trees and checking each one of them for completeness appears to be the only way to do it [33]. It is not that hard to check if in a given tree (or in any of its equivalent trees), there exists a node whose left and right subtrees are exactly identical. We discuss that in the following section.

5.4 Tree Reduction

From the definition of completeness in Chapter 2, Section 3.2.1 it follows that if we have a tree in which a node has identical left and right subtrees, then we can replace the subtree with that node as root, with its left or right subtree. The new tree is guaranteed to have a lower cost (and at least as efficient in its classification of containers) as compared to the original tree. However, the tree might still be incomplete because of absence of certain types of sensor(s) or because of having the same problem with at some other node. If we forget the problem of missing sensor types, and keep removing all redundant nodes, we will reach a point in which no node would have its left and right subtrees identical. However, the tree can still have an equivalent tree which might have the same problem. If we make sure that all its equivalent trees are "complete", the tree must be in its minimal or "irreducible form". Therefore,

Irreducible Trees. A tree will be called irreducible, if in every tree that belongs to its cost-equivalent class of trees, at any non-leaf node in the tree, its left and right sub-trees are not identical.

There exists an efficient algorithm for reducing any given binary decision tree into its minimal or irreducible form. Please see [12] for details of the algorithm.

5.4.1 Canonical Representation of an Equivalence Class

In order to exploit the shrinkage of tree space due to transposition-equivalence of trees, we need a unique, canonical representation of an equivalence class. With a unique, canonical form of every equivalence class, we only need to check the cost of only one of the trees of that class. Section 5.5.1 describes how we incorporate the use of canonical form representation into our modified search algorithms. Now, to represent the entire equivalence class of trees, with a unique, canonical form, we adopt a lexicographic representation of the equivalence class. To obtain this lexicographic arrangement in the given irreducible tree, we find all sensors that appear in all the paths from the root node to leaf nodes of the given tree. Then, by using repeated transposition operations, we pull the smallest sensor out of those sensors up as the root node. Then we repeat the

same procedure recursively on the left and right subtrees of the root node and proceed until the lexicographical arrangement is achieved. The resultant tree then represents the entire equivalence class uniquely. Any tree in the equivalence class can be converted to the same canonical representation using the above mentioned procedure. Figure 5.8 shows an example of converting a given tree irreducible tree to its unique canonical representation.

5.5 Space of Equivalence Classes of Irreducible Trees

Following from the above discussion on the definition of monotonic trees, it is quite evident that with our current constraint of identical threshold for every instance of a sensor type, a non-monotonic tree, in fact, can be an optimum tree. Therefore, it seems logical to include all non-monotonic trees in our search space of trees. Since it is very hard to verify if a given tree satisfies the definition of completeness as defined in Section 5.3, we therefore decided not to enforce equi-completeness. However, since for any given tree, the irreducibility constraint discussed above can be verified quite efficiently without actually enumerating all its equivalent trees, we decided to impose this restriction on the trees in our search space. The irreducibility constraint also partly takes care of the completeness constraint by checking that no two subtrees of a node should be identical. We still have to additionally check that all the sensor types appear at least once in the irreducible form of any given tree. Therefore, after making all these changes, our search space is now modified to a space of equivalence classes of irreducible trees with atleast one instance of every sensor type. Also, each equivalence class in this space is represented by a unique canonical form tree.

5.5.1 Searching through the New Space

In this section we modify the search algorithms discussed in Chapter 4 to enable them to search for the optimum trees in the new space of equivalent classes of irreducible trees. We have to make only a few changes to both the stochastic search method and the genetic algorithms based method to make them suitable for our new space

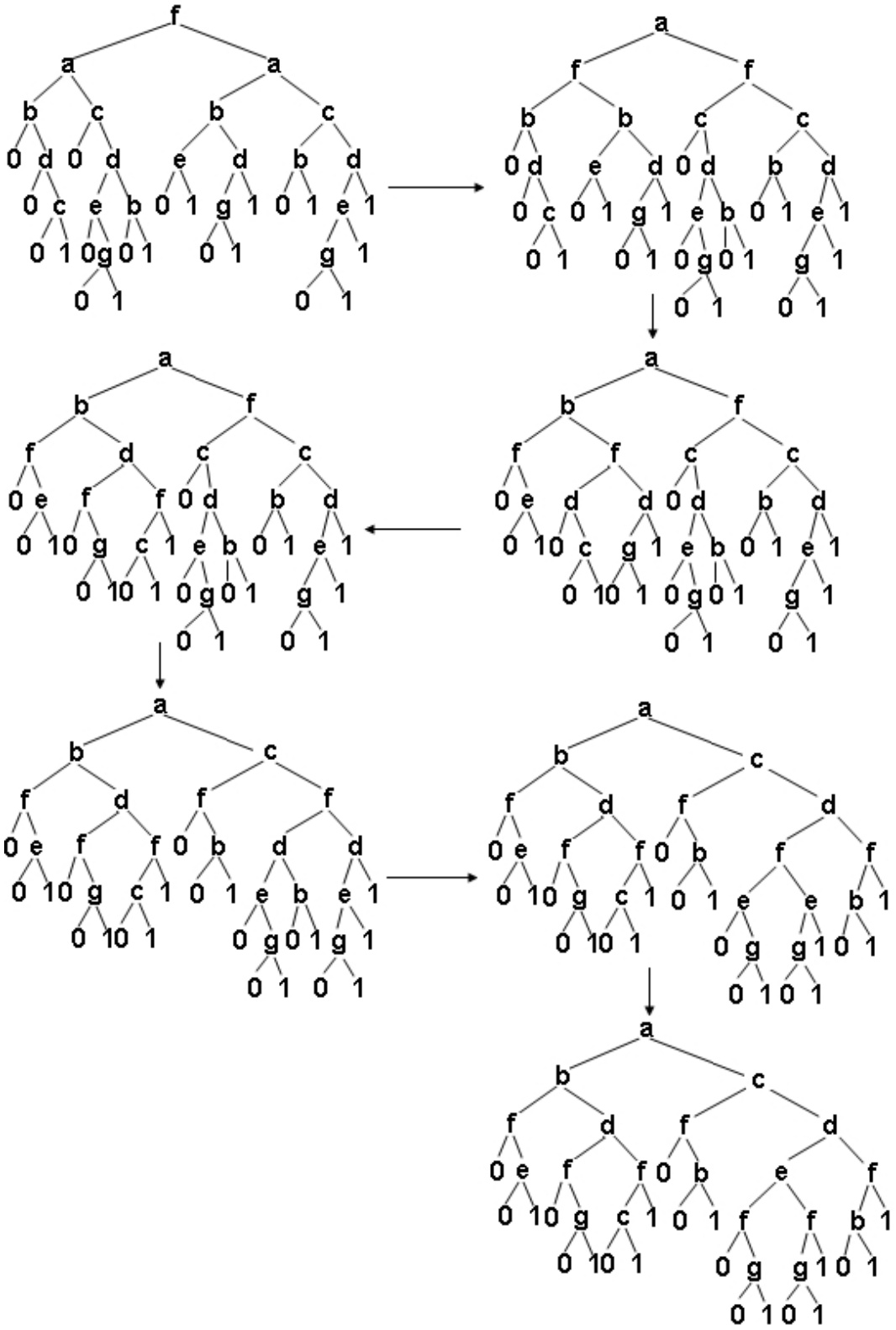


Figure 5.8: An example showing the procedure for obtaining unique canonical representation of a given tree.

of trees. Firstly, we need to relax the monotonicity constraint on the output trees in the definitions of the four neighborhood operations. Secondly, while generating a random tree or generating trees with the crossover and mutation operations, the generation of monotonic and non-monotonic trees as output should be equally likely. Lastly, all the trees generated (either randomly, through neighborhood operations or through mutation and crossover operations) should be reduced to their canonical form before checking for their cost. This would facilitate us in following two ways. 1. We can get the minimum cost of the tree after removing any redundancies. 2. We do not need to check more than one tree belonging to the same equivalence class for their costs. Apart from these changes, the rest of the search algorithms remain entirely the same. The results from these algorithms on the new space of equivalent classes of irreducible trees are presented in Chapter 6.

Chapter 6

Experimental Results

In this chapter we present results from all the experiments from Chapters 2, 4 and 5. For all the experiments, the model parameters were assumed as prescribed by Stroud and Saeger [31]. Most of the experiments were performed for 3,4 and 5 sensors. Some of the results are compared to those of Anand et. al. [1].

6.1 Optimizing Thresholds

Our first set of experiments focused on evaluating the optimization algorithm for threshold setting that we proposed in Chapter 2, Section 2.4. In these experiments, for any given tree, starting with some vector of sensor thresholds, we tried to reach a minimum cost in as few steps as possible. For comparison purposes, we did an exhaustive search for optimum thresholds with a fixed step size in a broad range for 3 and 4 sensors. Also, in all these experiments, the various sensor parameter values were kept the same as in the threshold variation experiments conducted in [1]. Both the misclassification costs and the prior probability of occurrence of a "bad" container were fixed as the respective averages of their minimum and maximum values used by Anand et. al. in [1]. We did this for both the method of exhaustive search over thresholds with fixed step size and the optimization method described in Chapter 2, Section 2.4.3, to maintain consistency throughout our experiments. With our new methods we were able reach a minimum every time with a modest number of iterations. For example, for 3 sensors, it took an average of 0.032 seconds, as opposed to 1.34 seconds using exhaustive search over thresholds with fixed step size, to converge to the minimum for all 114 trees using MATLAB on an Intel 1.66 GHz dual core machine with 1GB system memory. Similarly, for 4 sensors, it took an average of 0.195, seconds as opposed to 317.28 seconds using

exhaustive search, to converge to the minimum for all 66,600 trees. Figure 6.1 shows the plots for minimum costs for all 114 trees for 3 sensors using both the methods.

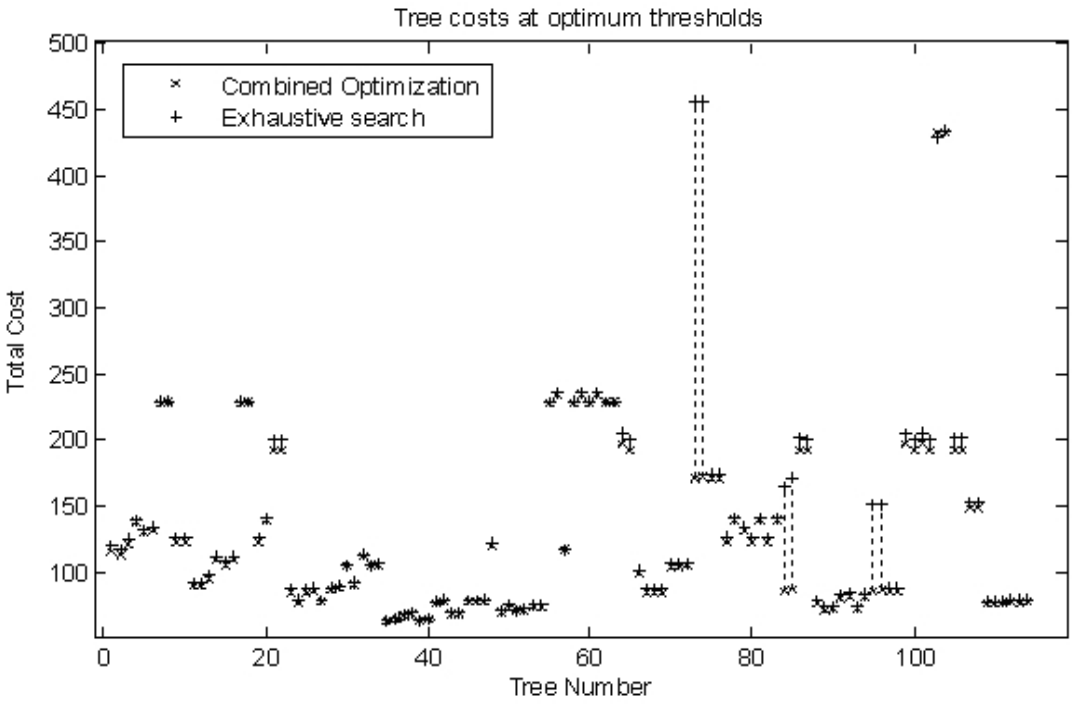


Figure 6.1: Minimum costs for all 114 trees for 3 sensors. To avoid confusion, dashed vertical lines join markers for the same trees.

In each case the minimum costs obtained using the optimization technique are equal to or less than those obtained using the exhaustive search. Also, many times the minimum obtained using the optimization method was considerably less than the one from the exhaustive search method.

6.2 Searching the CM Tree Space

Our second set of experiments consists of searching the tree space for the cheapest trees using the stochastic search method and the genetic algorithms based search method described in Chapter 4.

6.2.1 The Stochastic Search Method

For the stochastic search method, we randomly started 10 times at some tree in the CM tree space of 66,600 trees for 4 sensors and then kept moving stochastically in the neighborhood of the current tree, forming a chain of trees, until we reached a minimum. The exponent $\frac{1}{t}$ was initialized to 1 and was incremented by 1 (i.e., $\frac{1}{t'} = \frac{1}{t} - 1$) after every 10 hops in a chain. We found that the average number of trees evaluated for their costs for a set of 100 such experiments was 4890. Table 6.1 summarizes the results of these experiments. Each row in the table corresponds to the tree number that was obtained as the least cost tree along with its cost and frequency (out of 100). The last column in the table gives the rank of each of these tree minima among all the local minima in the entire tree space. For example, the algorithm was able to find the best tree 42 times, second best tree 15 times and so on. The algorithm was able to find one of the least cost trees most of the time. However, these trees are different from the lowest cost trees obtained in Anand et al. [1] and are in fact less costly than those trees. Another important observation is that although each of these four trees differ in structure, they still correspond to the same Boolean function, $F(\mathbf{abcd}) = 0001010101111111$, where the ith digit gives $F(\mathbf{abcd})$ for the ith binary string **abcd** if strings are arranged in lexicographically increasing order. Interestingly, this Boolean function is both complete and monotonic.

Tree Number[1]	Cost[2]	Frequency[3]	Mode Rank
30995	59.3364	42	1
30959	59.3364	15	2
31011	59.3364	25	3
31043	60.1924	10	4

[1] Tree numbers differ from those used in Anand et al. For actual tree structures, please see Figure 6.2 .

[2] The costs of the first three trees differ only in the 14th place after decimal, but all the trees are listed in the order of increasing costs.

[3] Frequency out of 100.

Table 6.1: Summary of results for stochastic search for 4 sensor tree space.

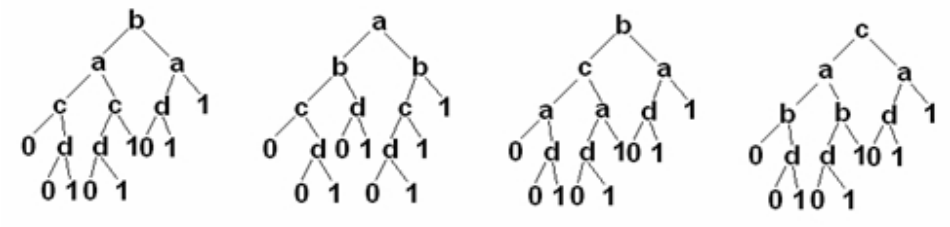


Figure 6.2: Best trees obtained for 4 sensors. Trees numbered 30995, 30959, 31011 and 31043 respectively.

6.2.2 Genetic Algorithms based Search Method

In our third set of experiments, we performed similar experiments using genetic algorithms described in Chapter 4, Section 4.2.2. For 4 sensors, we started with a random population of 20 trees. At each crossover step we crossed every tree in this population with every other tree. We set the value of $k = 1$ so that we get one new tree for each crossover operation. Also, with $m = 3$, we performed the mutation step after every three generations. During every mutation step, we replaced one half of the population of best trees ($M = 2$) with random samples from the tree space. We performed a set of 100 such experiments each consisting of a total of 27 generations (including the ones obtained after mutations). We noticed that for each such experiment, we had to evaluate only 1439.6 trees for their costs on an average. Table 6.2 summarizes the results of these experiments. It is clear from the results that every time we were able to find one of the cheapest trees in the CM tree space. Also, we observed that as opposed to the stochastic search technique, where the algorithm returned a single best tree in most of the cases, the genetic algorithm based search algorithm returned a whole population of trees, most of which belonged to the 50 cheapest trees.

6.2.3 Going beyond 4 Sensors

Our algorithms enable us to go beyond 4 sensors. We performed experiments for up to $n = 10$. We present the results for $n = 5$ and $n = 10$. The sensor parameter for the fifth sensor were assumed to be the average of those of first four sensors. The last five sensors were assumed to be exactly identical to the first five sensors. For example:

Tree Number[1]	Cost[2]	Frequency[3]	Mode Rank
30995	59.3364	52	1
30959	59.3364	40	2
31011	59.3364	8	3

[1] Tree numbers differ from those used in Anand et al. For actual tree structures, please see Figure 6.2 .

[2] The costs of the first three trees differ only in the 14th place after decimal, but all the trees are listed in the order of increasing costs.

[3] Frequency out of 100.

Table 6.2: Summary of results for genetic algorithm based search for 4 sensor tree space.

sensor **f** has same parameters as sensor **a**, sensor **g** has same parameters as sensor **b** and so on. (Obviously all the ten sensors can be set to different threshold values). As we commented earlier also, that the number of trees grow double exponentially, therefore for $n = 5$, the CM tree space consists of more than 22.5 billion trees. Due to better performance of genetic algorithms (GA) based search method over stochastic search method, we were motivated to use it for obtaining the low-cost trees. However, we modified it a little bit in a sense that instead of starting just once with a big random population, we started multiple times with smaller random populations. Also, the total number of generations were also not fixed apriori. Instead, we stopped producing new trees, if the best population remained constant over a few subsequent generations. We then performed GA on all the optimum trees obtained from each such start until the cost of the best trees stabilizes again. We noticed that for $n = 5$, with 100 runs, the results were on the same lines as those for $n = 4$, since we got only very few trees as the best trees, with similar costs. Figure 6.3 gives the actual structures of these trees and their respective cost.

For $n = 10$, we performed only a few runs. We observed that unlike $n = 4, 5$, here we always ended up with different population of best trees. However, the cost of these trees were close and also, the trees were similar at the top few nodes. Figure 6.4 gives the actual structures of these trees and their respective cost.

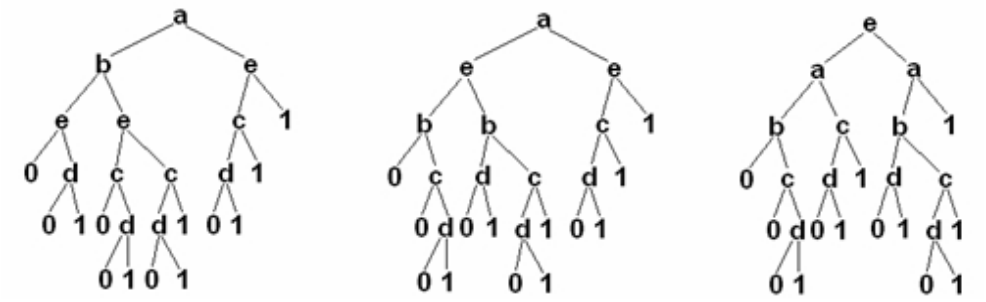
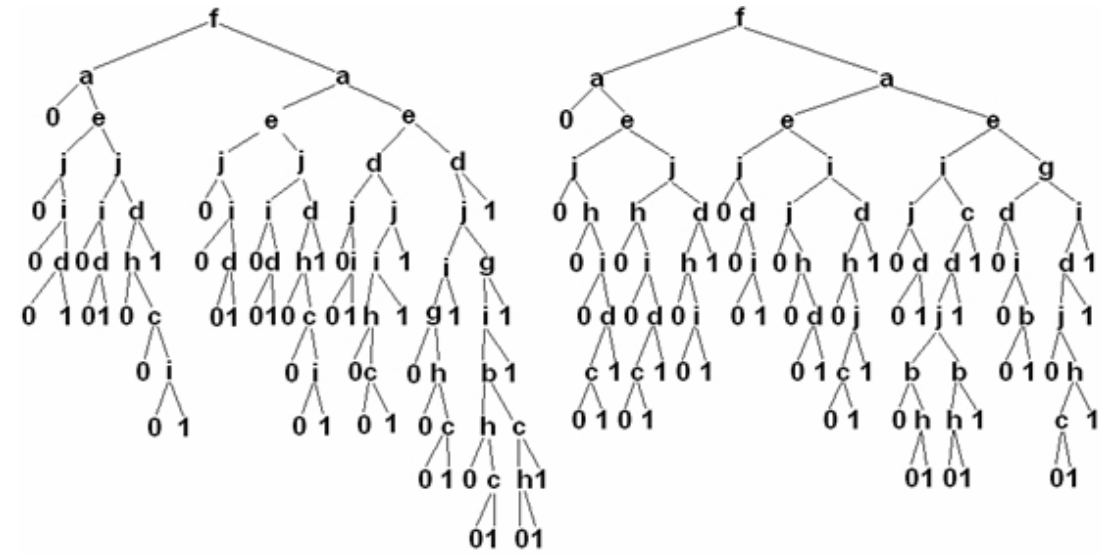


Figure 6.3: Best trees obtained for 5 sensors over 100 runs. The cost of each of these trees is 41.4668.



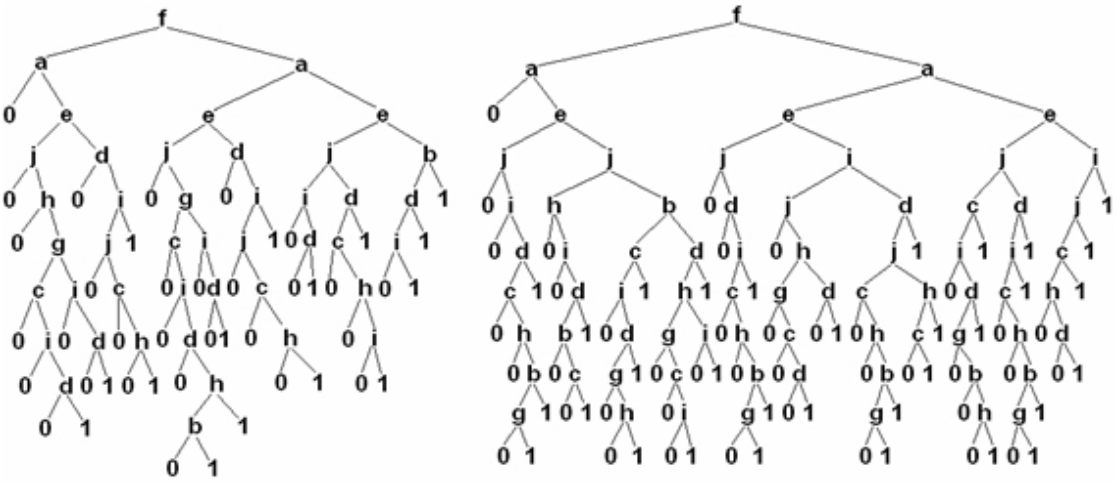


Figure 6.4: Best trees obtained for 10 sensors over four different runs. Their costs are 8.6508, 8.5499, 8.7236 and 8.6189 respectively.

6.3 Searching the Space of Equivalence Classes of Irreducible Trees

This section presents results for Chapter 5. As discussed in Chapter 5, we modify the search space of trees by removing the monotonicity constraint and introducing the concept of equivalence classes of trees. We then modify the search algorithms to suit the newer space of trees. The following sections present results obtained using the stochastic search method and the genetic algorithms based method.

6.3.1 The Stochastic Search Method

As discussed in Chapter 5, we modify the neighborhood operations to allow generation of non-monotonic trees. For example, the *split* operation is now allowed to insert a new non-monotonic sensor in a branch where that sensor is not already present. Also, each neighboring tree obtained is converted to its "irreducible form" before calculating its cost. However, the lexicographical canonical representation of a tree is only stored to make sure that none of the other trees belonging to the same equivalence class is checked for its cost. We again started randomly 20 times with some tree in the search space of trees for 4 sensors and then kept moving stochastically in the neighborhood of the current tree, forming a chain of trees, until we reached a minimum. The exponent $\frac{1}{t}$ was initialized to 1 and was incremented by 1 (i.e., $\frac{1}{t'} = \frac{1}{t} - 1$) after every 10 hops in a chain. We found that the average number of trees evaluated for their costs for a set of 100 such experiments was 9805. The algorithm was able to find a tree from the cheapest equivalence class 88 times. Interestingly, in all these runs, the best classes of trees obtained constitued the trees that came out as best trees in the experiments presented in earlier sections (Sections 6.2.1 and 6.2.2).

6.3.2 Genetic Algorithms based Search Method

The genetic algorithms based search method discussed in Chapter 4, Section 4.2.2 is modified to suit the new space of equivalence classes of trees and to include the non-monotonic nature of the trees. To do this, we modify the crossover and mutation operations so that the generation of monotonic and non-monotonic trees is equally likely.

Like in the stochastic search method discussed in Section 6.3.1, each tree is converted to its irreducible form before checking for its cost, while the lexicographic canonical form is just stored to make sure that multiple trees belonging to same equivalence class are not checked for their cost. The rest of the algorithm remains entirely the same. For 4 sensors, we conducted 100 experiments. In each experiment we started 20 times with a random population of 6 different trees. However, the total number of generations were not fixed apriori. Instead, we stopped generating new trees, if the best population remained constant over a few subsequent generations. We then performed GA on all the optimum trees obtained from each of the 20 starts until the cost of the best trees stabilizes again. We observed that for $n = 4$, the algorithm was able to find the cheapest cost tree 92 out of 100 times. We noticed that for every such experiment, we had to evaluate around 5008 trees for their costs on an average. The cheapest cost tree class obtained constitutes the cheapest trees obtained in the experiments in Section 6.2.2 (the first three trees in Figure 6.2). Similarly, for $n = 5$, 84 out of 100 times, the cheapest cost tree class returned by the algorithm constitutes the cheapest trees obtained in Section 6.2.2.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

After analysing the binary decision trees, their various spaces and their equivalence classes, we draw the following conclusions. Firstly, we noticed that the exhaustive search method, for finding the optimum thresholds for a given tree, become practically infeasible beyond a very small number of sensors. The various threshold optimization techniques discussed in our work provide faster and better ways to calculate the optimal total cost of a tree. Secondly, we noticed that the exhaustive search method, for finding the cheapest tree in the entire space of trees is also hard to extend beyond a very small number of sensors.

Thirdly, we noticed that expanding the ideas of monotonicity and completeness from Boolean decision functions as suggested by Stroud and Saeger [31] to binary decision trees, proved to be a more reasonable for two reasons. Certain trees obtained from incomplete/ non-monotonic Boolean decision functions are potentially valid BDTs and, it facilitates tree search algorithms. This way we get rid of the exhaustive search method over the entire space of trees. Further, we were able to prove that the proposed space of complete and monotonic trees is irreducible under the defined neighborhood operations.

We described a couple of efficient search methods to find the best trees in the space of complete and monotonic BDTs. Later, we were able to extended these ideas to the space of equivalent classes of decision trees. These search methods (especially the genetic algorithms based search method) helps us go beyond trees with four sensor types successfully.

Finally, we discussed the ideas of completeness and monotonicity more thoroughly

and argued why certain non-monotonic trees can actually be the cheapest cost trees. This led us to removing the monotonicity constraint from the tree space. We introduced the ideas of tree equivalence and tree reduction that helped us shrink the tree space. We also discussed a way to convert any given binary decision tree to its unique, canonical form.

7.2 Future Work

As one of the tasks for future work, a more basic and rigorous analysis of monotonicity is required. The empirical results show that the best trees do not change even after lifting the monotonicity constraint. As per our present understanding, the reason for this behavior relates to the specific sensor parameter values assumed in our experiments. In other words, non-monotonic trees can appear as the cheapest trees for some different values of sensor parameters; a more detailed analysis is needed.

Another important future task could be the removal of the constraint that every instance of a sensor in a tree should be set to identical thresholds. This would lead to the expansion of dimensionality of the threshold optimization function from the number of sensor types to the number of actual sensor instances in a tree and hence would make the task of finding the optimal thresholds for any given tree slower. This might also limit the present search capabilities to smaller trees. Therefore, newer or more optmized versions of the current search algorithms might be required.

References

[1] S. Anand, D. Madigan, R. Mammone, S. Pathak, and F. Roberts. Experimental analysis of sequential decision making algorithms for port of entry inspection procedures. In *S. Mehrotra, D. Zeng, H. Chen, B. Thuraisingham, and F-X Wang (eds.), Intelligence and Security Informatics, Proceedings of ISI-2006, Lecture Notes in Computer Science # 3975, Springer-Verlag, New York*, 2006.

[2] M. N. Azaiez and V. M. Bier. Optimal resource allocation for security in reliability systems. *European Journal of Operational Research*, 181(2), 2007.

[3] Y. Ben-Dov. Optimal testing procedures for special structures. *Management Science*, volume 27:1410–1420, 1981.

[4] H. Binnendijk, C. C. Leigh, T. Coffey, and H. S. Wynfield. The virtual border: Countering seaborne container terrorism. *Defence Horizons*, August 2002.

[5] E. Boros, E. Elsayed, P. Kantor, F. Roberts, and M. Xie. Optimization problems for port-of-entry detection systems. In *Intelligence and Security Informatics: Techniques and Applications, H. Chen and C. C. Yang (eds), Springer*, to appear.

[6] E. Boros, L. Fedzhora, P.B. Kantor, K. Saeger, and P. Stroud. Large scale lp model for finding optimal container inspection strategies. Technical Report RRR26-2006, Rutgers Center for Operations Research, Rutgers University, October 2006.

[7] R. Butterworth. Some reliability fault testing models. *Operations Research*, volume 20:335–343, 1972.

[8] H. A. Chipman, E. I. George, and R. E. McCulloch. Bayesian cart model search. *Journal of the American Statistical Association*, 93(443):935–947, 1998.

[9] H. A. Chipman, E. I. George, and R. E. McCulloch. Extracting representative tree models from a forest. *Working Paper July 98-07, Department of Statistics and Actual Science, University of Waterloo*, 1998.

[10] J. R. B. Cockett. The theory of discrete decision processes. Technical Report UT-CS-84-58, Department of Computer Science, University of Tennessee, October 1984.

[11] J. R. B. Cockett. Discrete decision theory: Manipulations. *Theoretical Computer Science*, 54(2-3):215–236, Oct 1987.

[12] J. R. B. Cockett and J. A. Herrera. Decision tree reduction. *JACM: Journal of the ACM*, 37, 1990.

[13] L. Cox, S. Chiu, and X. Sun. Least-cost failure diagnosis in uncertain reliability systems. *Reliability Engineering and System Safety*, 54:203–216, 1996.

[14] L. Cox, Y. Qiu, and W. Kuehner. Heuristic least-cost computation of discrete classification functions with uncertain argument values. *Annals of Operations Research*, 21:1–30, 1989.

[15] S. O. Duffuaa and A. Raouf. An optimal sequence in multicharacteristics inspection. *Journal of Optimization Theory and Applications*, 67(1):79–86, 1990.

[16] H. Fang and D. O'Leary. Modified cholesky algorithms: A catalog with new approaches. Technical Report CS-TR-4807, University of Maryland, August 2006.

[17] Z. Fu. A computational study of using genetic algorithms to develop intelligent decision trees. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1382–1387, 2001.

[18] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, 1981.

[19] J. Halpern. Fault-testing for k-out-of-n system. *Operation Research*, 22:1267–1271, Nov/Dec 1974.

[20] J. Halpern. The sequential covering problem under uncertainty. *INFOR*, 15:76–93, Feb 1977.

[21] Hyafil and Rivest. Constructing optimal binary decision trees is NP-complete. *IPL: Information Processing Letters*, 5, 1976.

[22] I. H. Lavalle and P. C. Fishburn. Equivalent decision trees and their associated strategy sets. In *Theory and Decision*, volume 23, pages 37–63, 1987.

[23] J. M. Loy and R. G. Ross. Global trade: America's achilles' heel. *Defense Horizon*, 7, 2002.

[24] D. Madigan, S. Mittal, and F. Roberts. Efficient sequential decision-making algorithms for container inspection operations. *Working paper. Rutgers University, New Brunswick NJ*, 2007.

[25] D. Madigan, S. Mittal, and F. Roberts. Sequential decision making algorithms for port of entry inspection: Overcoming computational challenges. In *Intelligence and Security Informatics*, pages 1–7. IEEE, 2007.

[26] R. Miglio and G. Soffritti. The comparison between classification trees through proximity measures. *Computational Statistics & Data Analysis*, 45(3), 2004.

[27] T. D. Nielsen and F. V. Jensen. Representing and solving asymmetric decision problems. *International Journal of Information Technology and Decision Making*, 2(2):217–263, 2003.

[28] A. Papagelis and D. Kalles. Breeding decision trees using evolutionary techniques. In *Proc. 18th International Conf. on Machine Learning*, pages 393–400. Morgan Kaufmann, San Francisco, CA, 2001.

[29] S. Pathak. Sensitivity analysis of tree-structured decision making algorithms for diagnostic applications. Master's thesis, Rutgers University, 2006.

[30] H. A. Simon and J. B. Kadane. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence*, 6(3):235–247, 1975.

[31] P. D. Stroud and K. J. Saeger. Enumeration of increasing boolean expressions and alternative digraph implementations for diagnostic applications. In *Proceedings Volume IV, Computer, Communication and Control Technologies*, pages 328–333, 2003.

[32] H. Zantema. Decision trees: Equivalence and propositional operations. In *Proceedings 10th Netherlands/Belgium Conference on Artificial Intelligence (NAIC'98) (November 1998), H. L. Poutr'e and J. van den Herik, Eds., pp. 157 – 166. Extended version appeared as report UU-CS1998 -14, Utrecht University.*, 1998.

[33] H. Zantema and H. L. Bodlaender. Finding small equivalent decision trees is hard. *International Journal of Foundations of Computer Science*, 11(2):343–354, 2000.

[34] H. Zhang, C. Schroepfer, and E. A. Elsayed. Sensor thresholds in port-of-entry inspection systems. In *Proceedings of the 12th ISSAT International Conference on Reliability and Quality in Design*, pages 172–176, 2006.