Efficient Sequential Decision-Making Algorithms for Container Inspection Operations

David Madigan,[1] Sushil Mittal,[2] Fred Roberts[3]

[1] *Department of Statistics Columbia University, New York*

[2] *Department of Electrical and Computer Engineering Rutgers, The state university of New Jersey*

[3] *DIMACS Rutgers, The State university of New Jersey*

Received 1 May 2009; revised 22 May 2011; accepted 26 May 2011
DOI 10.1002/nav.20472
Published online 2 August 2011 in Wiley Online Library (wileyonlinelibrary.com).

Abstract: Following work of Stroud and Saeger (Proceedings of ISI, Springer Verlag, New York, 2006) and Anand et al. (Proceedings of Computer, Communication and Control Technologies, 2003), we formulate a port of entry inspection sequencing task as a problem of finding an optimal binary decision tree for an appropriate Boolean decision function. We report on new algorithms for finding such optimal trees that are more efficient computationally than those presented by Stroud and Saeger and Anand et al. 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. © 2011 Wiley Periodicals, Inc. Naval Research Logistics 58: 637–654, 2011

Keywords: sequential decision making; Boolean function; binary decision tree; container inspection; heuristic algorithms

1. INTRODUCTION

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. 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. Stroud and Saeger [26] looked at this as a sequential decision making problem and formulated it in an important special case as a problem of finding an optimal binary decision tree for an appropriate binary decision function. Anand et al. [1] reported an experimental analysis of the Stroud–Saeger method that led to the conclusion that the optimal inspection strategy is remarkably insensitive to variations in the parameters needed to apply the method.

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.

Correspondence to: D. Madigan (madigan@stat.columbia.edu)

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 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 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

[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. Replicates of a particular sensor type may improve performance but such replicates must be combined to produce a single zero or one.

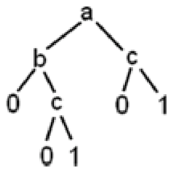

Figure 1. A binary decision tree τ with 3 sensors. The individual sensors classify good and bad containers towards left and right respectively.

(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).

Hyafil et al. [17] proved that even if the Boolean function F is fixed, the problem of finding the lowest cost BDT for it is hard (NP-complete). Brute force enumeration can provide a solution. However, even if the number of attributes, n, is as small as 4, this is not practical. Please recall that n refers to the number of sensor types and not to the total number of sensors present in a tree. 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 (called completeness and monotonicity—see below), Stroud and Saeger were unable to produce feasible methods for finding optimal BDTs beyond the case $n = 4$. They ranked all trees with up to four sensors according to increasing tree costs using a measure of cost we describe in Section 3. Anand et al. [1] described extensive sensitivity analysis showing that the Stroud–Saeger results were remarkably insensitive to wide-ranging changes in values of underlying parameters.

The purpose of this article is to describe computational approaches to this problem that are more efficient than those developed to date. We describe efficient approaches to the computation of sensor thresholds that seek to minimize the total cost of inspection. We also modify the special assumptions of Stroud and Saeger to allow search through a larger number of possible BDFs and introduce an algorithm for searching through the space of allowable BDTs that avoids searching through the Boolean decision functions entirely. Our experiments parallel those of Stroud and Saeger. This article is an expanded version of a short conference article by Madigan et al. [22], with added details and a detailed formal proof that our search methods in the larger space of allowable BDTs can reach any tree in the space from any other tree.

A variety of articles in recent years have dealt with the container inspection problem. Boros et al. [3] summarize a portion of this literature and Ramirez-Marquez [25] gives a more extensive survey of the literature. Dahlman et al. [9] provides an overview of the container security problem and present an outline of a potential comprehensive multilateral agreement on the use of containers in international trade. We close this section by reviewing the relevant literature.

A number of authors have built on the work of Stroud and Saeger [26]. One direction of work has been to study the sensitivity of optimal and near optimal trees to the input parameters used by Stroud and Saeger. As input parameters such as the costs of false positives and false negatives, the costs of delays, and so on, are estimated with more or less accuracy, one wants solutions whose sensitivity to changes in these parameters is known and tolerable. As noted above, Anand et al. [1] did an extensive sensitivity analysis of the Stroud–Saeger results and showed that the optimal inspection strategy is remarkably insensitive to variations in the parameters needed to apply the Stroud and Saeger [26] method. The article [22] introduces more efficient search heuristics that allow us to address problems involving more sensors, and it is the work of Ref. 22 that we expand on in this article.

In related research, Concho and Ramirez-Marquez [8,25] have used evolutionary algorithms to optimize a decision tree formulation of the inspection process. Their approach was based on the assumption that readings r_j by the jth sensor are normally distributed, with a different distribution depending on whether the container in question is "bad" or "good." Thresholds t_j were used to determine outcomes of inspections, with a container declared suspicious by the jth sensor if $r_j > t_j$. Here, the cost function used depends on the number of sensors used and the cost of opening a container for manual inspection if needed but does not take into account the cost of false positives or false negatives, which is a key feature of the work in Refs. 1, 26 and 22 and this article.

Another direction of work is to investigate the optimum threshold levels for sensor alarms so as to minimize overall cost as well as minimize the probability of not detecting hazardous material. Stroud and Saeger [26] developed threshold models in their work. We talk about some of this work here, building on [22]. For further related results, see Refs. 1, 3 and 10. Boros et al. [4] showed that multiple thresholds provide substantial improvement at no added cost. The problems of optimal threshold setting become much more complex and difficult to solve for a larger number of sensors. An alternative approach to determining threshold levels involves a simplifying assumption about the tree topology. Assuming a "series" topology (looking at one sensor at a time in a fixed order), one can first determine an optimal sequence of sensors. Once an optimum sequencing of sensors is obtained, the threshold level problem is then formulated. Zhang et al. [28] have used a complete enumeration approach to determine the optimum sequence of inspection stations and the corresponding sensors' threshold levels to solve problems with up to three sensors in series and parallel systems.

Elsayed et al. [10] studied specific topologies for inspection stations, specifically stations arranged in series or parallel topologies. They developed general total cost of inspection equations for n sensors in series and parallel configurations. In contrast to the work of Stroud and Saeger and that in this article, they disregarded costs of false positive and false negative classifications of containers.

Zhu et al. [29], in work extending [10], considered sensor measurement error independently from the natural variation in the container attribute values. They modeled situations when measurement errors exist (and are embedded) in the readings obtained by the inspection devices and used a threshold model to identify containers at risk for misclassification. They studied optimization of container inspection policies if repeated inspections of at-risk containers are a part of the process.

Boros et al. [4] extended the work of Stroud and Saeger and changed the formulation of the problem. Rather than minimizing expected cost determined as a combination of expected cost of utilizing an inspection protocol plus expected cost of misclassifying a container, they looked at the problem of maximizing the probability of detection of a "bad" container. They formulated a large-scale linear programming model yielding optimal strategies for container inspection. This model is based on a polyhedral description of all decision trees in the space of possible container inspection histories. The dimension of this space, while quite large, is an order of magnitude smaller than the number of decision trees. This formulation allowed them to incorporate both the problem of finding optimal decision trees and optimal threshold selection for each sensor into a single linear programming problem. The model can also accommodate budget limits, capacities, and so on, and one can solve it to maximize the achievable detection rate. Boros et al. were able to solve this model for four sensors, and branching that allows up to seven possibly different routing decisions at each sensor (in contrast to the binary routing solved by Stroud and Saeger, and implicit in Boolean models) in a few minutes of CPU time, on a standard desktop PC. They were also able to run the model for as many as seven sensors, when they allowed only binary decisions, as in Stroud and Saeger. It should be noted that Boros et al. also considered more container classifications than just the bad or good. They demonstrated the value of a mixed strategy applied to a fraction of the containers. Goldberg et al. [15] added budget constraints to the problem and considered the problem of finding an inspection policy that maximizes detection probability given that the cost of inspection cannot exceed a given budgeted amount.

Jacobson et al. [19] analyzed on baggage screening at airports and compared 100% screening with one type of screening device with screening with a second device when the first device says a bag is suspicious. They calculated costs and benefits of the two methods. (Jacobson et al. [20] also analyzed on baggage screening at airports and studied how integer programming models can be used to obtain optimal deployment of baggage screening security devices for a set of flights traveling between a given set of airports.)

The first step in the container inspection process actually starts outside the United States. To determine which containers are to be inspected, the US Customs and Border Protection (CBP) uses a layered security strategy. One key element of this strategy is the automated targeting system (ATS). CBP uses ATS to review documentation, including electronic manifest information submitted by the ocean carriers on all arriving shipments, to help identify containers for additional inspection. CBP requires the carriers to submit manifest information 24 h prior to a US-bound sea container being loaded onto a vessel in a foreign port. ATS is a complex mathematical model that uses weighted rules that assign a risk score to each arriving shipment in a container based on manifest information. The CBP officers then use these scores to help them make decisions on the extent of documentary review or physical inspection to be conducted [30]. This can be thought of as the first inspection test and the "sensor" is the risk-scoring algorithm. Thus, in some sense, all trees start with the first sensor and this sensor is then not used again. It is reasonable to think of more sophisticated risk scoring algorithms that also involve sequential decision making, going to more detailed analysis of risk on the basis of initial risk scoring results. The Canadian government uses similar methods. The Canadian Border Services Agency (CBSA) uses an automatic electronic targeting system to risk-score each marine container arriving in Canada. As with ATS, this Canadian system has several dozen risk indicators, and a score/weight for each indicator. The Canada Border Services Agency is applying a new performance metric, Improvement Curve, to measure risk-assessment processes at Canada's marine ports with improved efficiencies [**?**, 16]. Identifying mislabeled or anomalous shipments through scrutiny of manifest data is one step in a multilayer inspection process for containers arriving at ports described in Ref. 27. Other relevant work on risk scoring and anomaly detection from manifest data is found in Refs. 5 and 13.

2. COMPLETE, MONOTONIC BOOLEAN FUNCTIONS

The special assumptions Stroud and Saeger make to render computation more feasible are to limit consideration to so-called complete and monotonic Boolean functions. A Boolean function F is monotonic if given two strings $x_1x_2\dots x_n, y_1y_2\dots y_n$ with $x_i \geq y_i$ for all i, $F(x_1x_2\dots x_n) \geq F(y_1y_2\dots y_n)$. F is incomplete if it can be calculated by finding at most $n-1$ attributes and knowing the value of the input string on those attributes. For small values of n, Stroud and Saeger [26] enumerate all complete, monotonic Boolean functions and then calculate the least expensive corresponding BDTs under assumptions about various costs associated with the trees. Their method is practical for n up to 4, but not for $n = 5$. The problem is exacerbated by the number of BDFs. For example, for $n = 4$, there are 114 complete, monotonic Boolean functions and

11,808 distinct corresponding BDTs. By comparison, for unrestricted Boolean functions on four variables, there exist 1,079,779,602 BDTs! For $n = 5$, there are 6,894 complete, monotonic Boolean functions and 263,515,920 corresponding BDTs. Stroud and Saeger [26] showed that for the unrestricted case, the corresponding number of BDTs for $n = 5$ is $\sim 5 \times 10^{18}$.

3. COST OF A BDT

Following Anand et al. [1] and Stroud and Saeger [26], we assume that the cost of a binary decision tree is the total expected cost across potential outcomes. The overall cost comprises two components: (i) the expected cost of utilization of the tree and (ii) the expected cost of misclassification. The expected cost of utilization of a tree, C_{util}, is computed by performing a summation over the cost of using each sensor in the tree times the probability that a container is inspected by that particular sensor. We compute the expected cost of misclassification for a tree by calculating the probabilities of false positive (P_{FP}) and false negative (P_{FN}) misclassifications by the tree and multiplying by their respective costs C_{FP} and C_{FN}. Thus, the total cost C_{tot} is given by

$$C_{\mathrm{tot}} = C_{\mathrm{util}} + C_{\mathrm{FP}} * P_{\mathrm{FP}} + C_{\mathrm{FN}} * P_{\mathrm{FN}}.$$

Both costs (i) and (ii) depend on the distribution of the containers and the probabilities of misclassification of the individual sensors. For example, consider the decision tree τ in Fig. 1 with three sensors. The overall cost function to be optimized can be written as:

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

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 i given that the container is in state j, $i, j \in \{0, 1\}$. For real-valued attributes, Anand et al. [1] described a 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_{FN} and C_{FP} are the costs of a false negative and a false positive. (The notation here differs from that used by Anand et al. [1]) In the above expression, the first and second terms on the right-hand side together give the cost of utilization of the tree τ, whereas the third and fourth terms represent the costs of positive and negative misclassifications. For specific values of various costs and the parameters of the Gaussian model, please refer to Anand et al. [1] and Stroud and Saeger [26].

4. SENSOR THRESHOLDS

Sensors make errors. For sensors that produce a real-valued reading (e.g., Gamma radiation sensors), a natural approach to modeling sensor classification errors involves a threshold. With every sensor s, we associate a hard threshold, T_s. If the sensor reading for a container falls below T_s, then the output of that particular sensor in the tree is 0; it is 1 otherwise. The variation of sensor thresholds obviously impacts the overall cost of the tree. Although sensor characteristics are a function of design and environmental conditions, the thresholds can, at least in principle, be set by the decision maker. Therefore, mathematically, the optimum thresholds for a given tree τ can be defined as a vector of threshold values that minimizes the overall cost function $f(\tau)$ for that tree.

We model the design and environmental conditions by assuming that sensor values for good containers following a particular Gaussian distribution and sensor values for bad containers follow a different Gaussian distribution. This model was described in detail by Anand et al. [1] and Stroud and Saeger [26] along with approaches to finding optimal thresholds, based on assumptions about the parameters underlying the Gaussians. In particular, Anand et al. [1] described 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 $n = 4$, Anand et al. [1] reported 194,481 experiments leading to lowest cost trees, with the results being quite similar to those obtained in experiments by Stroud and Saeger [26]. Unfortunately, the methods do not scale and quickly become infeasible as the number of sensors increases.

One of the aims of this article 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. To accomplish this, we implemented various standard algorithms for nonlinear optimization. Numerical problems related to the calculation of the Hessian matrix $\mathbf{H}f(\tau)$ required for Newton's method led us to explore modified Cholesky decomposition schemes such as those described in Fang and O'Leary [11]. For example, a naïve way to convert a nonpositive definite matrix into a positive definite matrix is to decompose it to $\mathbf{LDL}^{\mathrm{T}}$ form (where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{D}$ is a diagonal matrix) and then make all the nonpositive 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 $\mathbf{LDL}^{\mathrm{T}}$ factorization method from

Gill et al. [14], which incorporates small error terms in both **L** and **D** at every step of factorization. Further, if the Hessian matrix $\mathbf{H}f(\tau)$ is ill-conditioned, we take small steps toward 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. Algorithm 1 summarizes the final scheme for finding the optimum thresholds.

Algorithm 1 A Combined Method for Optimum Threshold Computation

```
1.  Initialize $\mathbf{T_{start}}$ as a vector of random threshold values
2.  $\mathbf{T} \leftarrow \mathbf{inf}$
3.  while$|\mathbf{T} - \mathbf{T_{start}}| < 0.1\%$ of $\mathbf{T_{start}}$ do
4.      $\mathbf{T} \leftarrow \mathbf{T_{start}}$
5.      Compute $\partial\mathbf{f}$
6.      Compute $\mathbf{H}f(\tau)$
7.      if $\mathbf{H}\ f(\tau)$ is not positive definite, then
8.          Make $\mathbf{H}f(\tau)$ positive definite
9.      end if
10.     if $\mathbf{H}\ f(\tau)$ is well-conditioned, then
11.         $\mathbf{T_{start}} \leftarrow \mathbf{T_{start}} - [\mathbf{H}f(\tau)]^{-1}\partial\mathbf{f}$
12.     else
13.         $\mathbf{T_{start}} \leftarrow \mathbf{T_{start}} - \lambda\partial\mathbf{f}$
14      end if
15. end while
16. Output $\mathbf{T_{opt}} \leftarrow \mathbf{T}$
```

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.

5. SEARCHING THROUGH A GENERALIZED TREE SPACE

The previous section describes how we choose optimal sensor thresholds for a specific tree. We now discuss algorithms for searching tree space to find low-cost trees. First we fine-tune Stroud and Saeger's original definition of completeness and monotonicity to better suit the application.

5.1. Revisiting Completeness and Monotonicity

As noted in Section 2, Stroud and Saeger [26] limit their analysis to complete, monotonic Boolean functions. However, as we shall illustrate below, incomplete and/or nonmonotonic Booleans functions can in fact lead to useful trees (trees that represent viable inspection strategies). We propose here definitions of monotonicity and completeness for trees themselves rather than the Boolean functions whence the trees derive. We show that some incomplete and/or nonmonotonic Boolean functions can sometimes lead to complete and monotonic trees. We shall study a class of trees called CM

a b c	F(abc)
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

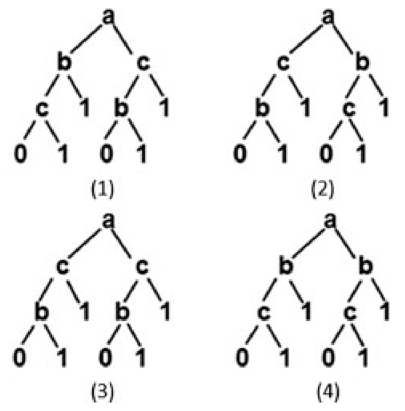

Figure 2. A Boolean function incomplete in sensor **a**, and the corresponding decision trees obtained from it.

trees, showing that it is much larger than the class of BDTs corresponding to complete, monotone Boolean functions, yet allows for efficient search algorithms that lead to very low cost trees consistently. By no means do we assert that there are no other useful trees than CM trees. Consider, for example, the Boolean function F and its corresponding BDT's shown in Fig. 2. 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. Sensor **a** is redundant in trees (iii) and (iv). Such considerations lead to the following definition:

5.1.1. *Complete Decision Trees*

A binary decision tree will be called complete if every sensor type (attribute) occurs at least once in the tree and, at any nonleaf node in the tree, its left and right subtrees are not identical.

Next consider the Boolean function and BDT's in Fig. 3. The Boolean function is not monotonic—when $\mathbf{b} = 1$ and $\mathbf{c} = 0$, $\mathbf{a} = 0$ yields an output of 1 whereas $\mathbf{a} = 1$ yields

a b c	F(abc)
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

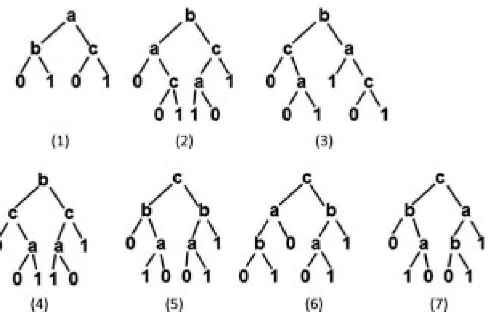

Figure 3. A Boolean function nonmonotonic in sensor **a**, and the corresponding decision trees obtained from the function.

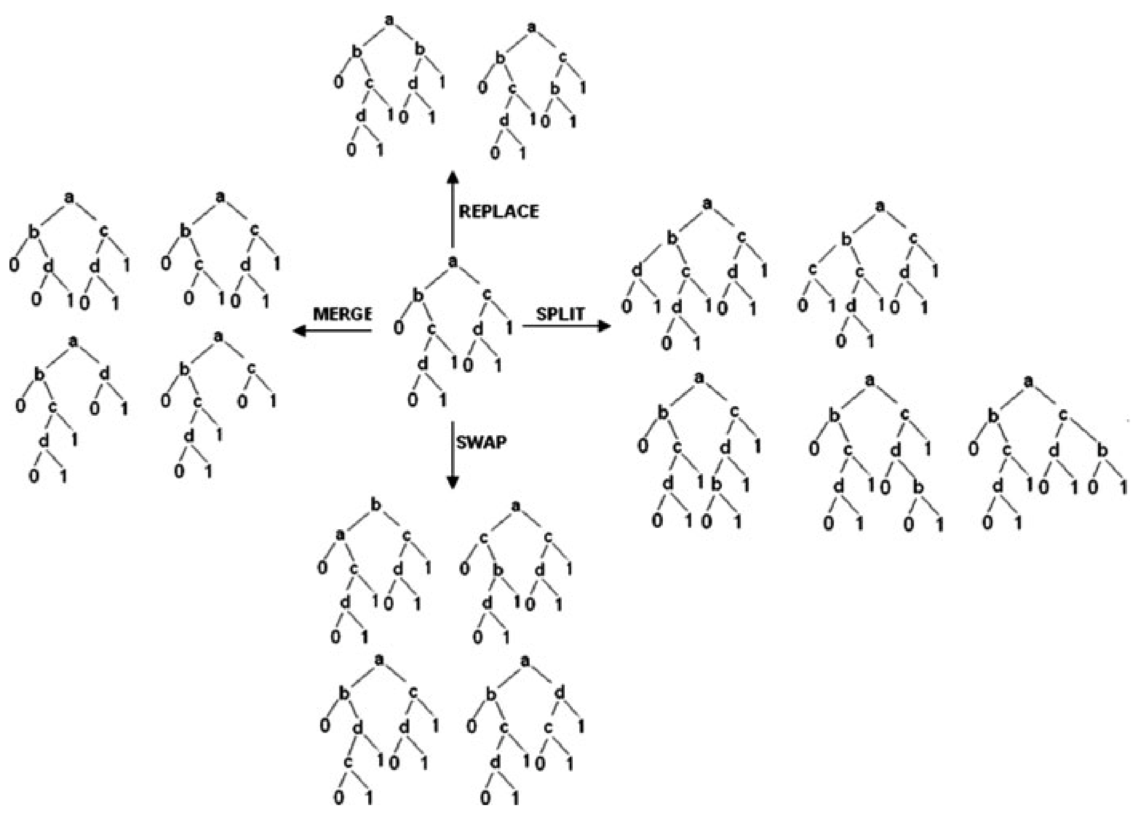

Figure 4. An example to illustrate the notion of neighborhood.

an output of 0. Except for tree (i), the corresponding trees also exhibit this nonmonotonicity, because there is a right arc from **a** to 0 or a left arc from **a** to 1 or both. However, tree (i) has no such problems and might well be a useful tree. Thus, we have the following definition:

5.1.2. Monotonic Decision Trees

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.

It is straightforward to show that:

- all BDT's corresponding to complete Boolean functions are complete,
- all BDT's corresponding to monotonic Boolean functions are monotonic, and
- the number of complete and monotonic trees increases very rapidly with the increasing number of sensors. There exist 114 complete, monotonic binary trees with three sensors and 66,600 with four sensors.

5.2. Tree Neighborhood and Tree Space

As shown in Stroud and Saeger [26], the number of binary decision trees corresponding to complete, monotonic Boolean functions increases exponentially with the addition of each new sensor. Expanding the space of trees in which to search for a cost-minimizing tree to the space of complete, monotonic trees, CM tree space, actually increases the number of possible trees but can decrease the computational challenge. We propose here a heuristic search strategy that builds on notions of neighborhoods in CM tree space.

Chipman et al. [6, 7] and Miglio and Soffritti [23] provide a comparison of various definitions of neighborhood and proximity between trees. Chipman et al. [7] 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. (Figure 4 gives an example of neighboring trees obtained from these operations for a particular tree.)

- Split: Pick a leaf node, 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 nonleaf 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 child, collapse both of them and shift the subtree up in the tree by one level. The nodes on which both these operations are performed are selected in such a fashion that the resulting trees are 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.

It is easy to show that these moves take a tree in CM tree space into another tree in CM tree space. Appendix II presents a proof that these moves generate an irreducible process in CM tree space. That is, for any pair of trees τ_1 and τ_2 in CM tree space, there exists a finite sequence of operations selected from the four operations above that start at τ_1 and end atτ_2 In fact, the "replace" operation is not needed for this proof but is useful in the search algorithm.

5.3. Tree Space Traversal

5.3.1. The Stochastic Search Method

We have explored alternate ways to exploit these operations to search for a tree with minimum cost in the entire CM tree space. Our initial approach 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 nine modes in the entire CM space of 114 trees for three sensors and 193 modes in the space of 66,600 trees for four sensors. To address the problem of getting stuck in a local 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 h hops until we reach a minimum. Specifically, if the algorithm is at a particular tree, τ, then the probability of moving to a particular neighbor τ' is given by:

$$P_{\tau\tau'} = c(f(\tau)/f(\tau'))^{1/t},$$

where $f(\tau)$ and $f(\tau')$ are the costs of trees τ and τ' and c is the normalization constant. Therefore, as the temperature is decreased, the probability of moving to the least expensive tree in the neighborhood increases. Algorithm 2 summarizes the stochastic search algorithm.

Algorithm 2 Stochastic Search Method using Simulated Annealing

```
1.  for $p = 1$ to numberOfStartPoints do
2.     $t \leftarrow 1$
3.     numberOfHops $\leftarrow 0$
4.     currentTree $\leftarrow$ random(allTrees)
5.     do
6.        Compute $f(\tau)$
7.        neighborTrees $\leftarrow$ findNeighborTrees
            (currentTree)
8.        for all $\tau' \in$ neighborTrees
9.           Compute $f(\tau')$
10.          Compute $P_{\tau\tau'}$
11.       end for
12.       currentTree $\leftarrow$ random(neighborTrees, $\mathbf{P}_{\tau\tau'}$)
13.       numberOfHops $\leftarrow$ numberOfHops $+ 1$
14        if numberOfHops $= h$ then
15.          $t \leftarrow t - \Delta t$
16.          numberOfHops $\leftarrow 0$
17.       end if
18.    while $f(\tau) > f(\tau') \forall \tau' \in$ neighborTrees
19. end for
20. Output lowest cost tree over all $p$
```

5.3.2. 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. With reference to 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 in the Refs. 2, 12, 24. Also, Im et al. [18] and Li et al. [21] describe applications where genetic and evolutionary algorithms were used to solve highly multimodal problems.

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 inτ_i)

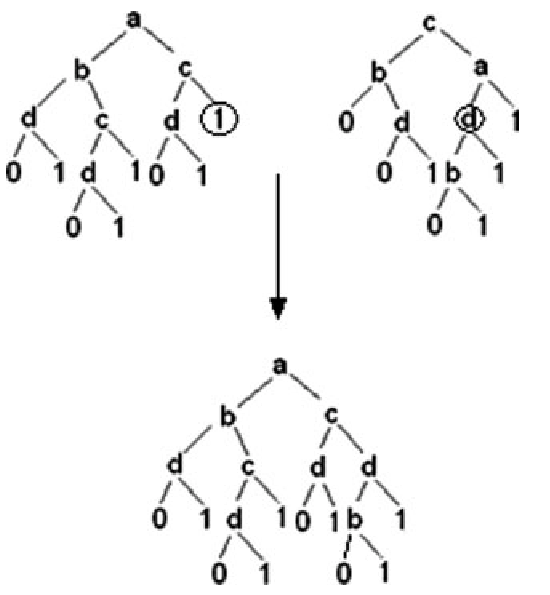

Figure 5. An example of a crossover between two trees.

with $\tau_{js'_1}$ (rooted at s'_1 in τ_j). A typical crossover operation is shown using the example in Fig. 5:

For every pair of trees, such random crossover operations are performed repeatedly until we get a specified number, N_{co}, of distinct trees or have exhausted all possible crossover operations. All the trees thus obtained are then put in the gene pool. 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 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 g_{mut} generations of the algorithm. We do two types of mutations. The first type consists of generating all the neighboring trees of the current best population of trees using the four operations used in the stochastic search method and putting these trees into the gene pool. The second type of mutation operation consists of replacing a fraction, $1/M$ $(M > 1)$ of N, the total number of 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 current gene pool. Algorithm 3 summarizes the genetic algorithm-based search algorithm.

Algorithm 3 Genetic Algorithms based Search Method

```
1.  Initialize $bestPop \leftarrow$ **generateTreesRandomly**$(N)$
2.  Initialize $genePool \leftarrow bestPop$
3.  Initialize $lastMutation \leftarrow 0$
4.  **for** p = 1 to $totalNumberOfGenerations$ **do**
5.    **for all** $\tau_i, \tau_j \in bestPop, i \neq j$
6.      $GATrees \leftarrow$ **generateGATreesRandomly**
          $(\tau_i, \tau_j, N_{\text{co}})$
7.      $genePool \leftarrow genePool \cup GATrees$
8.    **end for**
9.    $bestPop \leftarrow$ **selectBestTrees**$(genePool, N)$
10.   $lastMutation \leftarrow lastMutation + 1$
11.   **if** $lastMutation = g_{\text{mut}}$ **then**
12.     **for all** $\tau \in bestPop$ **do**
13.       $neighborTrees \leftarrow$ **findNeighborTrees**$(\tau)$
14.     $genePool \leftarrow genePool \cup neighborTrees$
15.     **end for**
16.     $bestPop \leftarrow$ **selectBestTrees**$(genePool, N)$
17.     $bestPop \leftarrow$ **selectBestTrees**$(bestPop, N - N/M)$
18.     $bestPop \leftarrow bestPop \cup$
          **generateTreesRandomly**$(N/M)$
19.     $genePool \leftarrow genePool \cup bestPop$
20.     $lastMutation \leftarrow 0$
21.   **end if**
22. **end for**
23. Output $bestPop$
```

6. EXPERIMENTAL RESULTS

6.1. Optimizing Thresholds

Our first set of experiments focused on evaluating the optimization algorithm for the threshold setting that we proposed in Section 4. In these experiments, for any given tree, starting with some vector of sensor thresholds, we tried to reach a minimum cost by adjusting thresholds 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 three and four sensors. Also, in all these experiments, the various sensor parameter values were kept the same as in the threshold variation experiments conducted in Anand et al. [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. [1]. To maintain consistency throughout our experiments, we did this for both the method of exhaustive search over thresholds with fixed step size and the optimization method described in Algorithm 1. With our new methods, we were able reach a minimum every time with a modest number of iterations. For example, for three sensors, it took an average of 0.032 s, as opposed to 1.34 s 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 1-GB system memory. Similarly, for four sensors, it took an average of 0.195 s, as opposed to 317.28 s using exhaustive search, to converge to the minimum for all 66,600 trees. Figure 6 shows the plots for minimum costs for all 114 trees for three sensors using both the methods. 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 CM Tree Space: The Stochastic Search Method

Our second set of experiments considered the stochastic tree search algorithm proposed in Section 5.3.1. These experiments were conducted on the CM tree space of 66,600 trees for $n = 4$. Each experiment was started 10 times from some randomly chosen CM tree, moving stochastically in the neighborhood of the current tree, until a locally minimum cost tree was found. The exponent $1/t$ was initialized to 1 and was incremented by 1 after every 10 hops. The outcome of the experiment was the tree with minimum cost from all the trees visited in the 10 runs. The average number of trees visited per experiment (averaged over 100 replications of the experiment). Table 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 true best tree 42 times, true second best tree 15 times and so on. Thus, 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. Also, interestingly, this Boolean function is both complete and monotonic.

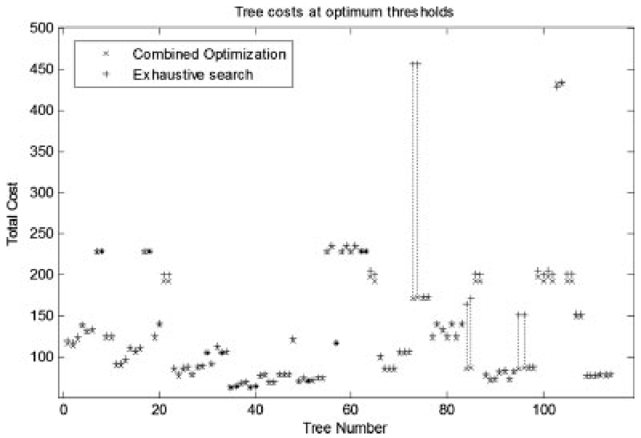

Figure 6. Minimum costs for all 114 trees for three sensors. To avoid confusion, dashed vertical lines join markers for the same tree.

Table 1. Summary of results for stochastic search for four sensor tree space.

Tree number[a]	Cost[b]	Frequency[c]	Mode rank
30995	59.3364	42	1
30959	59.3364	15	2
31011	59.3364	25	3
31043	60.1924	10	4

[a]Tree numbers differ from those used in Anand et al. [1]. For actual tree structures, please see Figure 13 in Appendix III.
[b]The costs of the first three trees differ only in the 14th place after the decimal, but all the trees are listed in the order of increasing costs.
[c]Frequency out of 100.

6.3. Searching CM Tree Space: Genetic Algorithm-Based Search Method

We performed similar experiments using the genetic algorithm described in Section 5.3.2. For $n = 4$, 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 $N_{\text{co}} = 1$ so that we get one new tree for each crossover operation. Also, with $g_{\text{mut}} = 3$, we performed the mutation step after every three generations. During every mutation step, we replaced 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 observed that for each such experiment, we had to evaluate on average only 1439.6 trees for their costs. Table 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

Table 2. Summary of results for genetic algorithm based search for four sensor tree space.

Tree number[a]	Cost[b]	Frequency[c]	Mode rank
30995	59.3364	52	1
30959	59.3364	40	2
31011	59.3364	8	3

[a]Tree numbers differ from those used in Anand et al. [1]. For actual tree structures, please see Figure 13 in Appendix III.
[b]The costs of the first three trees differ only in the 14th place after the decimal, but all the trees are listed in the order of increasing costs.
[c]Frequency out of 100.

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 cheapest 50 trees. Figure 7a shows the histogram of the actual costs of the trees found for $n = 4$. Figure 7b shows the zoomed-in version of the left-tail of the same histogram with the costs of the 20 cheapest trees found overlain in dotted vertical lines.

6.4. Going beyond four Sensors

We performed experiments for up to $n = 10$ sensors. Here, we present the results for $n = 5$ and $n = 10$. The sensor parameters for the fifth sensor were assumed to be the average of those of first four sensors. The last five sensors were assumed to be identical to the first five sensors; sensor **f** has the same parameters as sensor **a**, sensor **g** has same parameters as sensor **b** and so on. However, all 10 sensors can be set to different threshold values. For these larger-scale experiments we used the GA approach with multiple random restarts. In addition, rather than fixing the number of generations in advance, we ran the algorithms until the best population remained constant over several subsequent generations. We then performed GA on all the optimum trees obtained from each such start until the cost of the best trees stabilized again. For $n = 5$, with 100 runs, the GA converged on a small number of trees with similar costs. Please see Figure 14, Appendix III for actual structures of these trees and their respective cost. For $n = 10$, random restarts always ended up with different populations of best trees. However, the cost of these trees were close and also, the trees were similar at the top few nodes. Please see Figure 15, Appendix III for the actual structures of these trees and their respective costs. Also notice that even though for each n,the costs of the cheapest trees obtained are very close to each other, the trees themselves are not close according to the neighborhood measure adopted above.

7. DISCUSSION

As we have already noted, with binary decision trees, exhaustive search methods, both for finding the optimum thresholds for a given tree and for finding a minimum cost tree among all possible trees, become practically infeasible beyond a very small number of sensors. The various characterizations and algorithmic techniques discussed in this article provide faster and better methods to explore the search space and arrive at a minimum efficiently. We were able to obtain results for 10 sensors using the stochastic search method described above; results for even larger numbers of sensors are possible.

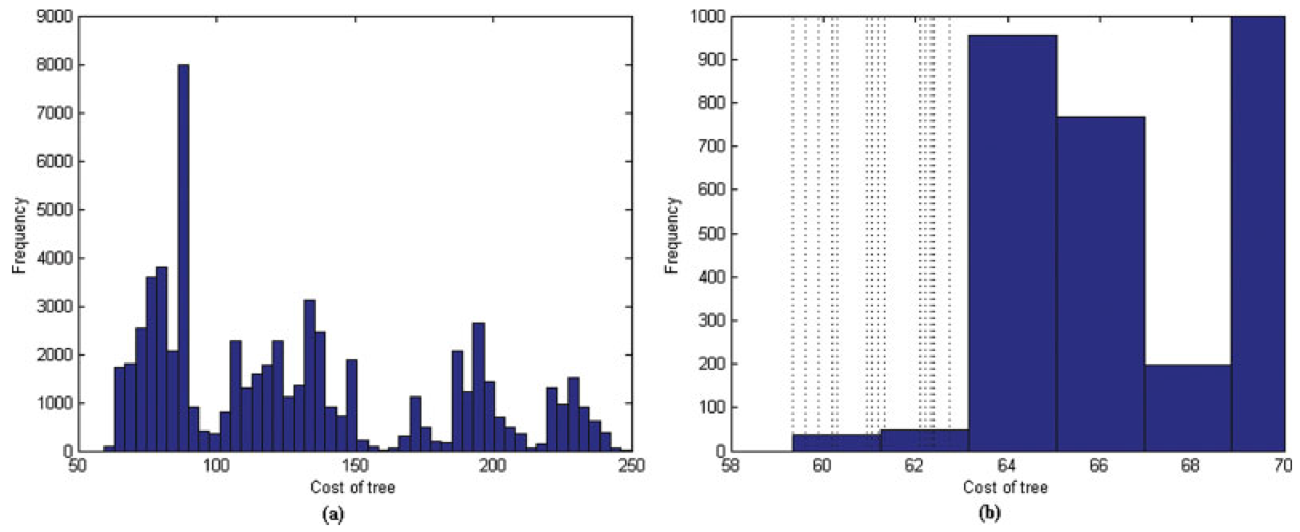

Figure 7. (a) Histogram of costs of all 66,600 trees for $n = 4$. (b). Left tail of the histogram. The dotted lines show the costs of 20 best trees found using the genetic algorithm-based search method. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

APPENDIX I: TERMINOLOGY

A rooted binary tree is a connected, directed, acyclic graph denoted by a pair (V, E) where V is a finite set of nodes and $E \subseteq \{(v, w) \in V \times V | v \neq w\}$ is a set of edges, (i.e., a set of ordered pairs of distinct nodes) such that there are exactly two edges going out of any internal node and exactly one edge coming into it. For any $(v, w) \in E$, we call w the "child" node of v and v the "parent" node of w. Nodes sharing the same parent are called "sibling nodes." v is called a descendent of u and u an ancestor of v if and only if the unique path from the root to v passes through u. The unique node in a tree with no parent node is called the "root node" while the nodes with no descendents are called "leaf nodes". The internal nodes together with the root node are called "nonleaf nodes". The "subtree" at a node v is the binary tree with its root at v. If u and w are left and right children of a node v, then the subtrees formed at u and w are called the left and right subtrees of v respectively. The left and right subtrees of any node are called sibling subtrees of each other. A binary decision tree (BDT) τ is a rooted binary tree where the nonleaf nodes correspond to specific sensors and the leaf nodes represent the final decision outputs.

Merging a node v in a tree corresponds to performing a merge operation on that node while subtree removal at a node v corresponds to replacing the subtree at v in the tree with a leaf node. A node v in τ is at level l if exactly l edges connect v and the root node of τ. Alternatively, v is said to be at level l of τ. A levelset $L(\tau, l)$ of a tree τ is the set of nodes in τ at level l. If $l_{\max}$ is the maximal level of τ, then the level $l_{\max} - 1$ is called the maximal nonleaf level of τ.

Let CM^n represents the space of complete and monotonic binary decision trees in n sensors. We consider neighborhood operations chosen from the set {*Split, Swap, Merge, Replace*} (though it turns out that we do not need Replace for the proof of the main theorem). Note that $\forall\tau \in \mathrm{CM}^n$, $\mathrm{o}(\tau) \in \mathrm{CM}^n$ for any operation o. Let $\mathrm{O} = \langle \mathrm{o}_1, \mathrm{o}_2, \ldots, \mathrm{o}_z\rangle$ represents a finite sequence of neighborhood operations where $\mathrm{O}(\tau) = \mathrm{o}_z(\mathrm{o}_{z-1}(\ldots(\mathrm{o}_1(\tau))))$ and z is a positive integer.

We define the following binary relation on a pair of trees $\tau_i, \tau_j \in \mathrm{CM}^n$:

$\tau_i \mapsto \tau_j \Leftrightarrow \exists$ a finite sequence of neighborhood operations $\mathrm{O} = \langle \mathrm{o}_1, \mathrm{o}_2, \ldots, \mathrm{o}_z\rangle$ such that $\tau_j = \mathrm{O}(\tau_i)$, or $\tau_i = \tau_j$.

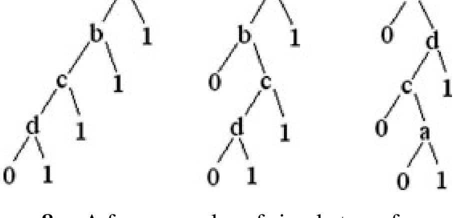

Figure 8. A few examples of simple trees for $n = 4$.

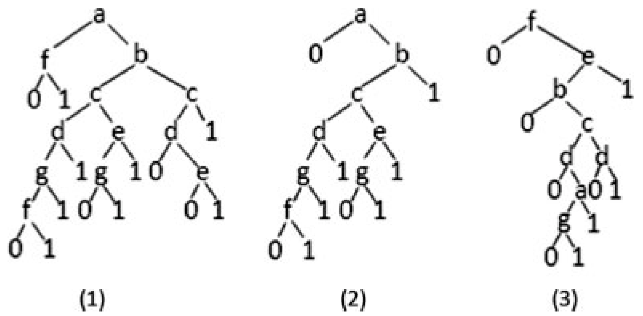

Figure 9. A few examples of partially simple trees. The first tree is partially simple to level 0, the second tree to level 2 and the third tree to level 3.

DEFINITION 1 (Simple tree): We define a simple tree $\sigma \subset \mathrm{CM}^n$ as a complete and monotonic binary decision tree such that the levelsets $L(\sigma, i)$ of τ, $i = 0, \ldots, n-1$, each contain exactly one nonleaf node. The unique path from the root node (i.e., level 0) to level $n-1$ containing all the nonleaf nodes is called the essential path. Figure 8 shows a few examples of simple trees for $n = 4$.

DEFINITION 2 (Partially simple tree): A partially simple tree to level l, σ_l, is defined as a complete and monotonic binary decision tree where the levelsets $L(\sigma_l, i)$ of σ_l, $i = 0, \ldots, l$, each contain exactly one nonleaf node. Figure 9 shows some examples of partially simple trees for $n = 7$.

APPENDIX II: PROOF OF CM TREE SPACE IRREDUCIBILITY FOR $N > 2$

To establish irreducibility, that is, that every tree in CM^n is obtainable from any other tree in CM^n by a sequence of neighborhood operations, we will first prove three lemmas that will form the backbone of the main proof. First, we will show that for every $\tau \in \mathrm{CM}^n$ there exists a simple tree $\sigma \in \mathrm{CM}^n$ such that $\tau \mapsto \sigma$. Next we will show that for every pair of simple trees, $\sigma, \sigma' \in \mathrm{CM}^n$, $\sigma \mapsto \sigma'$. Finally, we will show that for every $\tau \in \mathrm{CM}^n$ there exists a simple tree, $\sigma \in \mathrm{CM}^n$, such that $\sigma \mapsto \tau$.

LEMMA 1: For every tree $\tau \in \mathrm{CM}^n$ there always exists a simple tree $\sigma \in \mathrm{CM}^n$ such that $\tau \mapsto \sigma$ using only the neighborhood operations Split and Merge.

PROOF: We will first prove the following assertion:

Given any partially simple tree, $\sigma_l \in \mathrm{CM}^n$, there always exists a sequence of neighborhood operations O such that $\mathrm{O}(\sigma_l) = \sigma_{l+1}$, where σ_{l+1} is a partially simple tree to level $l+1$. The lemma will follow from this assertion, because we can then define n such sequences of operations $\mathrm{O}_1, \mathrm{O}_2, \ldots, \mathrm{O}_n$, such that $\mathrm{O}_n(\mathrm{O}_{n-1}(\ldots(\mathrm{O}_1(\tau))))$ is a simple tree. Otherwise, we consider a sequence of operations O,

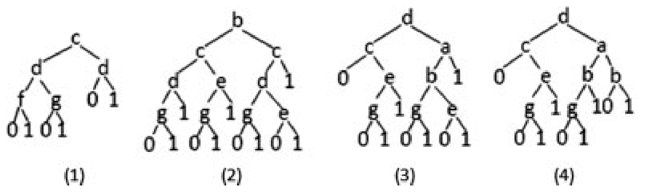

Figure 10. A few examples of trees to illustrate the selection criteria for subtree removal. In tree (1), we chose to remove the right subtree of sensor **c**, in tree (2), the right subtree of sensor **b**, in tree (3), the left subtree of sensor **d**, and in tree (4), the left subtree of sensor **d**.

which we will divide into two subsequences O^1 and O^2 such that $O(\sigma_l) = O_2(O_1(\sigma_l))$. O^1 will comprise zero or more Split operations and O^2 will comprise zero or more Merge operations. Let v_l be the sole nonleaf node at level l in σ_l. Therefore, both the left and right child nodes of v_l are nonleaf. We will proceed by retaining one of the subtrees of v_l and removing the other via a sequence of Merge operations. The selection of which subtree to remove is based on one of the following rules:

- If only one of the two subtrees of v_l is complete in $n - l$ sensors, then we choose to remove the incomplete one. Figure 10 tree (1) shows an example where we remove the right subtree of sensor **c** rather than the left one.
- If both the subtrees are complete in $n - l$ sensors, we choose to remove the one that has fewer nodes in it. Figure 10 tree (2) shows an example where we remove the right subtree of sensor **b** rather than the left one.
- If both the subtrees are incomplete in $n - l$ sensor types, then we choose to retain the subtree that has larger number of different sensor types in it. Figure 10 tree (3) shows an example where we remove the left subtree of sensor **d** rather than the right one.
- If both the subtrees are incomplete in $n - l$ sensor types and have an equal number of different sensor types, then we choose to retain the one that has fewer nodes in it. Figure 10 tree (4) shows an example where we remove the left subtree of sensor **d** rather than the right one.
- If both the subtrees are incomplete in $n-l$ sensor types and have equal number of different sensor types and equal number of nodes, we can merge any one of the two.

Notice that in cases 1 and 2, $O^1 = \emptyset$. In cases 3, 4, and 5, O^1 is defined as the sequence of Split operations performed iteratively, wherein each of the Split operations is performed at the maximal level node of the subtree that we decide to retain (choosing arbitrarily when there is more than one such node) until that subtree is complete in $n - l - 1$ sensors not present at levels 0 through l. Let $\sigma_l^1 = O^1(\sigma_l)$. Note that both σ_l and σ_l^1 are simple up to level l. Construction of O^2 is however nontrivial, because we cannot merge a node that would lead to a tree that is incomplete in a different node. For example, Fig. 11 shows an example of a tree where merger of the node **d** from the leftmost branch of the tree results in the tree becoming incomplete in a higher level node **b** (circled). Therefore, we make use of an algorithm called "*smartMerge*" to construct O^2. *smartMerge* guarantees that there always exists a node in the subtree that we want to remove, which can be removed (through a *Merge* operation) without making the resultant tree incomplete at any node.

***smartMerge*Algorithm**

Input: A partially simple tree σ_l^1.

Output: A partially simple tree σ_{l+1}.

Let v_l be the sole nonleaf node at level $l$in σ_l^1. Let τ_{sub1} and τ_{sub2} be the two subtrees of v_l in σ_l^1. Further, we assume without loss of generality that we want to retain τ_{sub2} and remove τ_{sub1}. Let the maximal nonleaf level of τ_{sub1} be m (as measured from the root node in τ_{sub1}). We first choose the nonleaf node v_{1m} at level m of τ_{sub1} (choosing arbitrarily when there is more than one such node) as the candidate node to merge. Note that if we merge v_{1m}, at most one of $m-1$ ancestor nodes of v_{1m} with level i, $i = 0, \ldots, m-2$ (again, $i = 0$ for the root node of τ_{sub1}) can render the resultant tree incomplete. In other words, there can be at most one of $m-1$ nodes in the resultant tree, whose left subtree would become exactly identical to the right subtree after we merge v_{1m}, thus resulting in an incomplete tree. As we always insert an appropriate leaf (0 if the node is a left node, 1 otherwise) after merging v_{1m}, the tree cannot become incomplete at the parent node of the new leaf. If a subtree (in τ_{sub1}) at a level r, $r \in \{0, \ldots, m-2\}$, denoted by τ_{r1} and containing v_{1m} becomes identical to its sibling subtree τ_{r2} after the merger of v_{1m}, then we cannot merge v_{1m}. Let v_{2m} be the sibling node of v_{1m} (obviously it would also be at level m). Anytime such a situation occurs, the next candidate node for removal is selected based on one of the following two possible configurations.

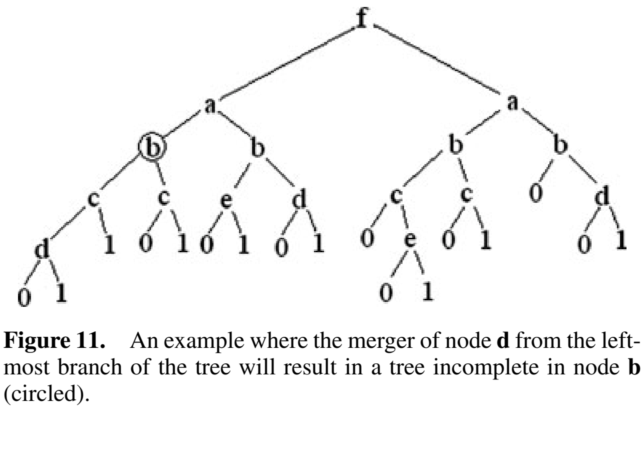

Figure 11. An example where the merger of node **d** from the leftmost branch of the tree will result in a tree incomplete in node **b** (circled).

1. v_{2m} is a nonleaf node: In this case, we propose to merge the exact counterpart of v_{2m}, denoted by v'_{2m}, in τ_{r2}. Again, we need to check at most $m-1$ nodes in the tree for completeness, but we know for sure that at least τ_{r1} cannot be identical to τ_{r2}. Therefore, there are just $m-2$ nodes that we need to check for completeness for the proposed merger of v'_{2m}. For example, in Fig. 12 tree **(1)**, $l=0$, and therefore, sensor **a** represents v_l. Let the left subtree of **a** be τ_{sub1} and the right subtree be τ_{sub2}. Further let sensor **f** (marked *) represent v_{1m}, where $m=4$. First we observe that if we remove sensor **f**, the tree cannot become incomplete in its parent node (sensor **d**). In fact, it would become incomplete in sensor **b** (circled). Therefore, as discussed above, we propose to remove sensor **g** present in the right subtree of sensor **b** (circled). This step of getting the new candidate node for removal, v'_{2m} from the previous one v_{1m} is shown as a transition from tree **(1)** to tree **(2)** in Fig. 12.
2. v_{2m} is a leaf node: Denote by u_{m-1} the parent node of v_{1m}. We propose to merge its counterpart u'_{m-1} in τ_{r2}. In this case, again we need to check $m-2$ nodes for completeness for the proposed merger of u'_{m-1}. For example, in Fig. 12 tree **(8)**, again $l=0$ and sensor **g** (marked *) represents v_{1m}, while its parent node (sensor d) represents u_{m-1}, where $m=4$. It is clear that if we remove sensor **g**, the tree would become incomplete in sensor **b** (circled). Also, as the sibling node of sensor **g** is a leaf node, therefore, we propose to remove the sensor **d** in the right subtree of sensor **b** (circled). This step of getting the new candidate node for removal u'_{m-1} from the previous one v_{1m} is shown as a transition from tree **(8)** to tree **(9)**.

Note that as this process continues, both the children of a candidate node might be nonleaf. For example, let us assume that v_{1p} $(m-g \le p \le m)$ is the proposed candidate node for removal after performing g candidate generation steps described above. At this point, there will be at most $m-g-1$ nodes to check for completeness for the proposed merger of v_{1p}. Further, assume that the merger of v_{1p} results in the tree becoming incomplete at a certain higher node at level s. Therefore, if v_{2p} is the sibling node of v_{1p}, we select its counterpart node v'_{2p} (with $m-g-2$ completeness constraints) in τ_{s2} as the next candidate node for merger. Let us assume that both the children of v'_{2p} are nonleaf nodes. In this case we try to merge a nonleaf node v'_{2q} at the maximal nonleaf level of the subtree rooted at v'_{2p}. If the level of that node in τ_{sub1} is q $(p < q \le m)$, then there are at most $m-g-2+(q-p)$ nodes to check for completeness for the proposed merger of that node. Therefore, even in the worst case, when $p=m-g$ and $q=m$, there are at most $m-g-2+(q-p) = m-g-2+(m-(m-g)) = m-2$ nodes to check for completeness. Thus we reduce by one the number of nodes that need to be checked. Therefore, by induction, we will reach to a node which requires $m-m=0$ nodes to be checked for completeness, and hence can be merged using the *Merge* operation. Then we repeat this procedure again to one of the nonleaf nodes at the maximal nonleaf level of the subtree that we want to merge, until σ_{l+1} is obtained. Algorithm 4 summarizes the *smartMerge* algorithm. Figure 12, trees **(1)** through **(16)**, show an example of obtaining a partially simple tree to level 1, σ_1, from an arbitrary tree in CM[7] (which is also trivially a partially simple tree to level 0). Further, trees **(16)** through **(23)** show how we can reach from the partially simple tree σ_1, to a simple tree just by repeated use of the *smartMerge* algorithm.

Algorithm 4 *smartMerge* Algorithm
```
0.  Input: A partially simple tree $\sigma_l^1$
1.    Initialize $v_{1m} \leftarrow$ a nonleaf node at the maximal
      nonleaf level of $\tau_{\text{sub1}}$
2.    while $\tau_{\text{sub1}}$ is not a leaf node, do
3.      flag_delete ← TRUE
4.      for $r = 0$ to $m-2$ level ancestors of $v_{1m}$, do
5.        if $\tau_{r1} = \tau_{r2}$, then
6.          if $v'_{2m}$ exists, then
7.            $m \leftarrow q$
8.            $v_{1m} \leftarrow v'_{2q}$
9.        else
10.           $m \leftarrow m-1$
11.           $v_{1m} \leftarrow u'_{m-1}$
12.         end if
13.         flag_delete ← FALSE
14.         break
15.       end if
16.     end for
17.     if flag_delete = TRUE
18.       merge $v_{1m}$
19.       $v_{1m} \leftarrow$ a nonleaf node at the maximal nonleaf
          level of $\tau_{\text{sub1}}$
20.     end if
21.   end while
22.   Output partially simple tree $\sigma_{l+1}$
```

Thus, we have shown that for any partially simple tree, σ_l, there exists a sequence of neighborhood operations (specifically, a series of zero or more Split operations followed by a sequence of zero or more Merge operations) that lead to a partially simple tree σ_{l+1} (i.e., a tree that is simple further down in the tree). As with n sensors, σ_{n-1} is a simple tree, and as every tree is partially simple to level 0, we have thus established the existence of a sequence of neighborhood

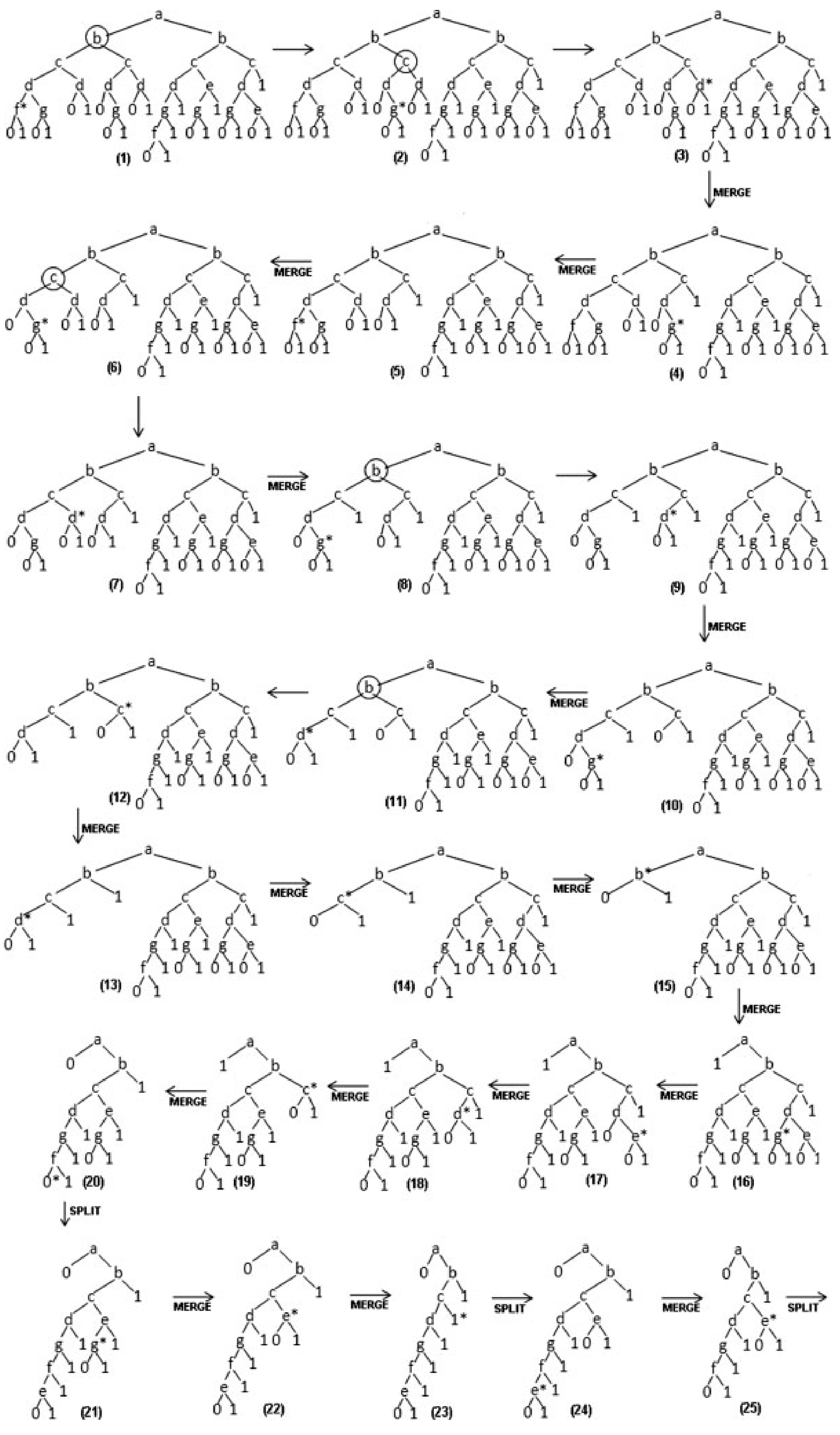

Figure 12. (Continued)

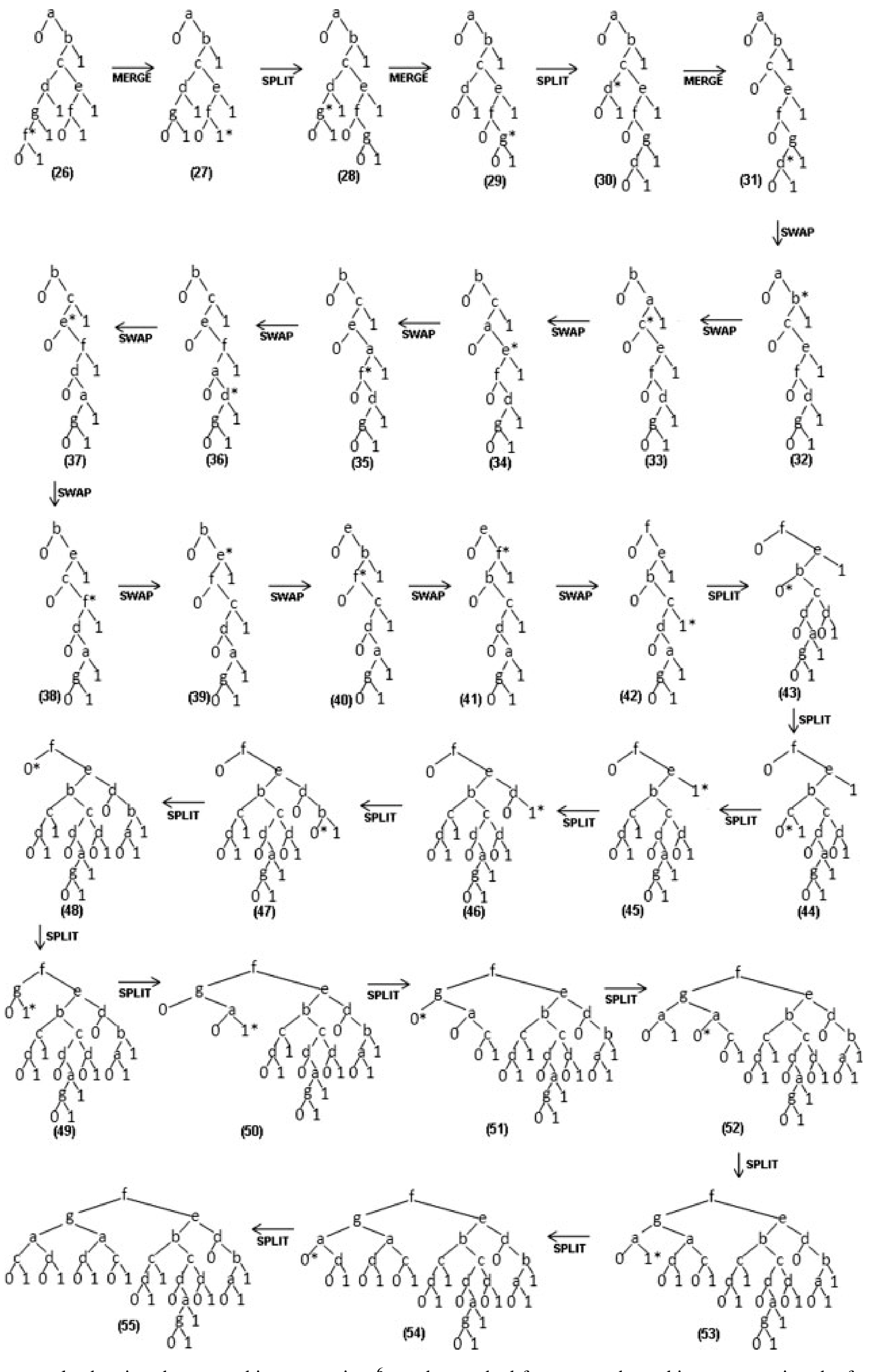

Figure 12. An example showing that any arbitrary tree in τ^6 can be reached from any other arbitrary tree using the four neighborhood operations repetitively. The node marked * in every tree is subject to a neighborhood operation while the nodes circled show a possible conflict with completeness constraint.

operations that starts with an arbitrary tree in CM^n and leads to a simple tree. This completes the proof of Lemma 1. □

LEMMA 2: For every pair of simple trees $\sigma, \sigma' \in$ CM^n, $\sigma \mapsto \sigma'$ using only the neighborhood operations Split, Merge, and Swap.

PROOF: We will prove that any simple tree σ' in CM^n can be reached from any other simple tree σ in CM^n using the four operations, repeatedly. Let P and P' be the essential paths of simple trees σ and σ' respectively, where:

$$P = v_0 \xrightarrow{d_1} v_1 \xrightarrow{d_2} \ldots\ldots \xrightarrow{d_{n-1}} v_{n-1}$$

$$P' = v_0' \xrightarrow{d_1'} v_1' \xrightarrow{d_2'} \ldots\ldots \xrightarrow{d_{n-1}'} v_{n-1}'$$

where $v_0, v_1, \ldots, v_{n-1}$ are the nonleaf nodes at level $0, 1, \ldots, n-1$ in the essential path of σ and $v_0', v_1', \ldots, v_{n-1}'$ are the nonleaf nodes at level $0, 1, \ldots, n-1$ in the essential path of σ'. Also, $D_1 = \{d_1, d_2, \ldots, d_{n-1}\}$ and $D_2 = \{d_1', d_2', \ldots, d_{n-1}'\}$ are direction $(n-1)$-tuples such that $d_i, d_i' \in \{Left, Right\}$, $i = 1, 2, \ldots, n-1$. We use $\bar{d_i}$ to denote the direction complementary to d_i, that is, $\bar{d_i} = Left$ iff $d_i = Right$ and vice-versa. Also, we say that $D_1 = D_2$ iff $d_i = d_i'$, $i = 1, 2, \ldots, n-1$. Lastly, by "adding v_i towards d at v_j", we mean inserting v_i as a child node of v_j (using the *Split* operation) where v_i is the left child when $d = Left$ and the right child when $d = Right$.

In order to go from σ to σ', we first modify σ so that $D_1 = D_2$. Then σ' can be obtained by one or more *Swap* operations. Let k be an integer such that $1 \leq k < n$ such that $\begin{Bmatrix} d_i = d_i' & \text{if } 1 \leq i < k \\ d_i \neq d_i' & \text{if } i = k \end{Bmatrix}$. If $k = n-1$, then D_1 differs from D_2 only in d_{n-1}. In this case we temporarily add v_{n-1} towards $\bar{d}_1$ at v_0. This can be done with a *Split* operation since in a simple tree, there is always a leaf node at each level, and therefore one at level 1. We then merge v_{n-1} from P, add v_{n-1} towards $\bar{d}_{n-1}$ at v_{n-2} (i.e., again using the *Split* operation) and finally merge $v_{n-1}$from $\bar{d}_1$ at v_0. If $k < n-1$, we insert v_{n-1} towards d_k' at v_{k-1} (because $\bar{d}_k = d_k'$) (this is the *Split* operation) and merge v_{n-1} from P. We then add v_{n-2} at v_{n-1} towards d_{k+1}' and merge v_{n-2} from P. We repeat this procedure for all $k \leq i < n-1$ until $D_1 = D_2$. After that we rearrange the nodes in the resultant tree using repeated *Swap* operations to obtain σ'. For example, in Fig. 12 let trees **(23)** and **(42)** be σ and σ' respectively in CM^7. Since $d_1 = d_1'$, $d_2 = d_2'$ and $d_3 \neq d_3'$, therefore $k = 3$. As discussed above, in tree **(24)**, we add sensor **e** towards the right of sensor **c** (v_2) and in tree **(25)**, we merge sensor **e** (v_6) from the left of sensor **f** (v_5). We then add sensor **f** towards the left of sensor **e** in tree **(26)** and merge sensor **f** from left of sensor **g** (v_4) in tree **(27)**. By proceeding in a similar fashion we can reach from tree **(27)** to tree **(31)**. Thereafter, by doing repeated *Swap* operations, we can reach from tree **(31)** to tree **(42)**. In this way, we prove that any simple tree can be reached from any other simple tree, using neighborhood operations repeatedly in CM^n. This completes the proof of Lemma 2. □

LEMMA 3: For any arbitrary tree $\tau' \in$ CM^n there exists a simple tree, $\sigma' \in$ CM^n, such that $\sigma' \mapsto \tau'$ using only the neighborhood operations Split and Merge.

PROOF: This lemma follows from the fact 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 we only merge nodes with both children as leaves, the converse is also true. Thus, we see that we can get from σ' to τ' using the steps to reach σ' from τ' in the exact reverse order. Fig. 12, trees **(42)** through **(55)**, provide an example of reaching to an arbitrary tree from a simple tree. Notice that all the steps in this sequence are reversible. This completes the proof of Lemma 3. □

THEOREM 1: In the space of complete and monotonic trees, every tree is reachable from every other tree by a sequence of neighborhood operations from the set {*Merge, Swap, Split*}.

PROOF: Lemmas 1, 2, and 3 give the result. □

APPENDIX III. TREE STRUCTURES

1. $n = 4$

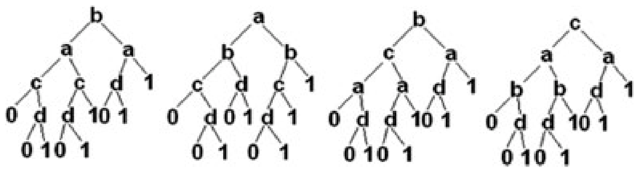

Figure 13. Some of the best trees obtained using the genetic algorithm based search method. The cost of each of the first three trees is very close to 59.3364 and that of the last one is 59.4150.

2. $n = 5$

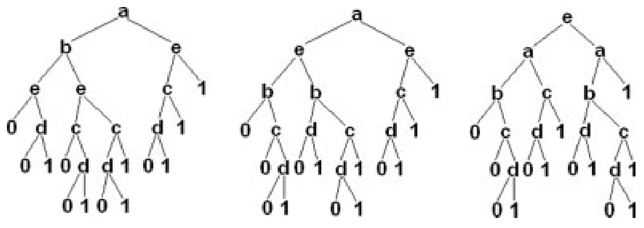

Figure 14. Best trees obtained over 100 runs. The cost of each of these trees is 41.4668.

3. $n = 10$

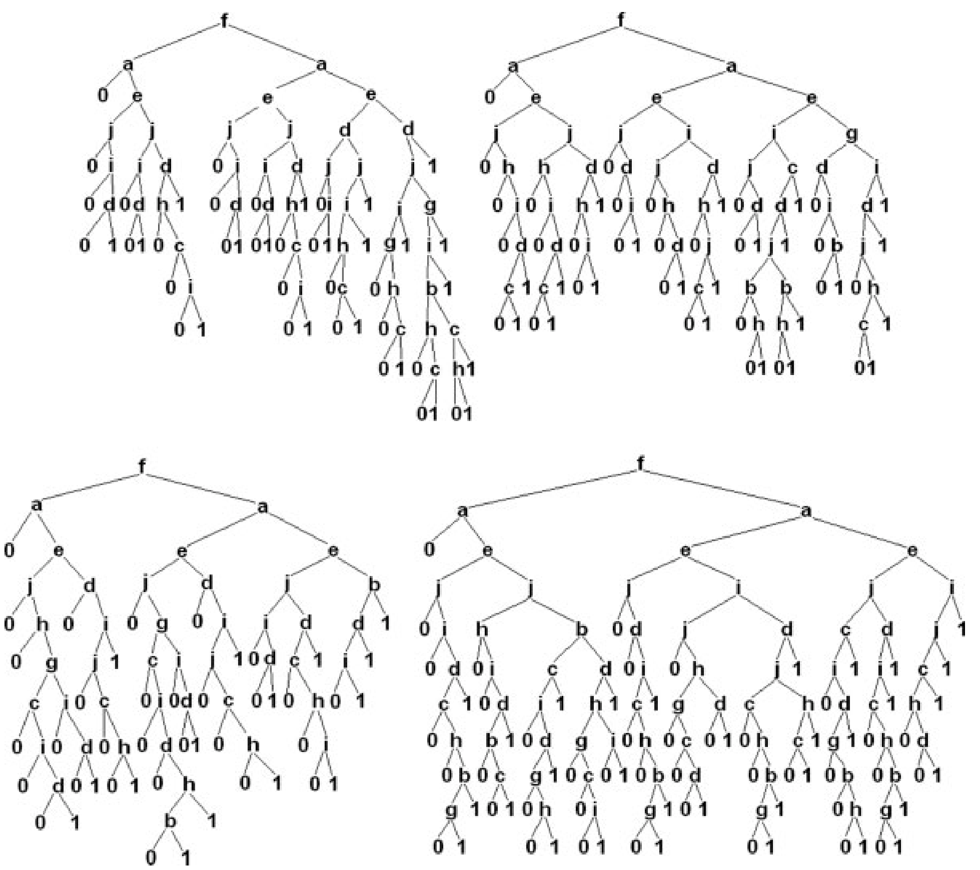

Figure 15. Best trees obtained for four runs. Their costs are 8.6508, 8.5499, 8.7236, and 8.6189 respectively.

ACKNOWLEDGMENTS

The authors thank Peter Meer and Oncel Tuzel for their ideas on implementing the Gradient Descent Method and Newton's Method for finding the optimum thresholds. They also thank Richard Mammone for many of the initial ideas that led to this research.

All three authors were supported by ONR grant number N00014-05-1-0237 and NSF grant number SES-0518543 and David Madigan was supported by grant DMS-0505599 to Rutgers University.

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 (Editors), Intelligence and security informatics, Proceedings of ISI, Lecture Notes in Computer Science #3975, Springer-Verlag, New York, 2006.

[2] Z. Bandar, H. Al-Attar, and D. McLean, "Genetic algorithm based multiple decision Tree Induction," in: Proceedings of the 6th International Conference on Neural Information Processing - ICONIP'99 – IEEE, Denver, Colorado, 1999, pp. 429–434, IEEE Cat. No. 99EX378. ISBN 0-7803-5871-6.

[3] E. Boros, E. Elsayed, P. Kantor, F. Roberts, and M. Xie, "Optimization problems for port-of-entry detection systems," H. Chen and C.C. Yang (Editors), Intelligence and Security Informatics: Techniques and Applications, Springer, Berlin, Germany, 2008, pp. 319–335.

[4] E. Boros, L. Fedzhora, P. Kantor, K. Saeger, and P. Stroud, A large scale LP model for finding optimal container inspection strategies, Naval Res Logistics Quart 56 (2009), 404–420.

[5] X. Chen, J. Cheng, and M. Xie, "A statistical approach for analyzing manifest data in pre-portal intelligence," working paper, DIMACS Center, Rutgers University, 2010.

[6] H.A. Chipman, E.I. George, and R.E. McCulloch, Bayesian CART Model Search, J Am Stat Assoc 93 (1998), 935–960.

[7] H.A. Chipman, E.I. George, and R.E. McCulloch, "Extracting Representative Tree Models from a Forest," Working paper 98-07, Department of Statistics and Actual Science, University of Waterloo, 1998.

[8] A. Concho and J.E. Ramirez-Marquez, An evolutionary algorithm for port-of-entry security optimization considering sensor threshold, Reliab Eng Syst Safety 95 (2010), 255–266.

[9] O. Dahlman, J. Mackby, B. Sitt, A. Poucet, A. Meerburg, B. Massinon, E. Ifft, M. Asada, and R. Alewine, Container security: A proposal for a comprehensive code of conduct, defense and technology paper, National Defense University Center for Technology and National Security Policy (2005).

[10] E. Elsayed, C. Schroepfer, M. Xie, H. Zhang, and Y. Zhu, Port-of-entry inspection: Sensor deployment policy and optimization, IEEE Trans Automation Sci Eng 6 (2009), 265–277.

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

[12] Z. Fu, A computational study of using genetic algorithms to develop intelligent decision trees, In: Proceedings of the 2001 Congress on Evolutionary Computation, Seoul, Korea, 2001.

[13] M.C. Ganiz, N.I. Lytkin, and W.M. Pottenger, "Leveraging higher order dependencies between features for text classification, Machine Learning and Knowledge Discovery in Databases," in: Buntine et al. (Editors), Lecture Notes in Computer Science, 5781, 2009, pp. 375–390.

[14] P.E. Gill, W. Murray, and M.H. Wright, Practical Optimization, Academic Press, New York, NY, 1981.

[15] N. Goldberg, J. Word, E. Boros, and P. Kantor, Optimal sequential inspection policies, RUTCOR Research Report 14-2008, Rutgers University, 2008.

[16] R. Hoshino, D. Coughtry, S. Sivaraja, I. Volnyansky, S. Auer, and A. Trichtchenko, Application and extension of cost curves to marine container inspection, Ann Oper Res (2009), 1–25.

[17] L. Hyafil and R.L. Rivest, Constructing optimal binary decision trees is NP-complete, Information Process Lett 5 (1976), 15–17.

[18] C. Im, H. Kim, H. Jung, and K. Choi, A novel algorithm for multimodal function optimization based on evolution strategy, IEEE Trans Magn 40 (2004), 1224–1227.

[19] S.H. Jacobson, T. Karnani, J.E. Kobza, and L. Ritchie, A cost-benefit analysis of alternative device configurations for aviation checked baggage security screenings, Risk Anal 26 (2006), 297–310.

[20] S.H. Jacobson, L.A. McLay, J.L. Virta, and J.E. Kobza, Integer programming models for deployment of airport baggage screening security devices, Optim Eng 6 (2005), 339–358.

[21] J.P. Li, M. Balazs, G. Parks, and P. Clarkson, A species conserving genetic algorithm for multimodal function optimization, Evol Comput 10 (2002), 207–234.

[22] D. Madigan, S. Mittal, and F.S. Roberts, "Sequential decision making algorithms for port of entry inspection: Overcoming computational challenges," G. Muresan, T. Altiok, B. Melamed, and D. Zeng (Editors), in: Proceedings of IEEE International Conference on Intelligence and Security Informatics, IEEE Press, Piscataway, NJ, 2007, pp. 1–7.

[23] R. Miglio and G. Soffritti, The comparison between classification trees through proximity measures, Comput Stat Data Anal 45 (2004), 577–593.

[24] A. Papagelis and D. Kalles, Breeding decision trees using evolutionary techniques, in: Proceedings of the Eighteenth International Conference on Machine Learning, Williamstown, MA, USA, 2001, pp. 393–400.

[25] J. Ramirez-Marquez, Port-of-entry safety via the reliability optimization of container inspection strategy through and evolutionary approach, Reliab Eng Syst Safety 93 (2008), 1698–1709.

[26] 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, Orlando, FL, USA, 2003, pp. 328–333.

[27] L. Wein, A. Wilkins, M. Bajeva, and S. Flynn, Preventing the importation of illicit nuclear materials in shipping containers, Risk Anal 26 (2006), 5.

[28] 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, Chicago, Illinois, USA, August 3–5, 2006, pp. 172–176.

[29] Y. Zhu, M. Li, C.M. Young, M. Xie, and E. Elsayed, Impact of measurement error on container inspection policies at port-of-entry, DIMACS Technical Report 2009-11, DIMACS Center, Rutgers University, Ann Oper Res (2010); DOI 10.1007/s10479-010-0681-6.

[30] United States Government Accountability Office (2006), Cargo container inspection, GAO-06-591T, GAO, Washington, DC, March 30, 2006.