Inferring Reduced Ordered Decision Graphs of Minimum Description Length

Arlindo L. Oliveira
IST/INESC
Lisbon, Portugal
aml@inesc.pt

Alberto Sangiovanni-Vincentelli
Dept. of EECS
UC Berkeley
alberto@ic.berkeley.edu

Abstract

We propose an heuristic algorithm that induces decision graphs from training sets using Rissanen's minimum description length principle to control the tradeoff between accuracy in the training set and complexity of the hypothesis description.

1 INTRODUCTION

Decision graphs can be viewed as a generalization of decision trees, a very successful approach for the inference of classification rules [Breiman et al., 1984, Quinlan, 1986]. The selection of decision graphs instead of decision trees as the representation scheme is important because many concepts of interest require very large decision trees. In particular, the quality of the generalization performed by a decision tree induced from data suffers because of two well known problems [Oliver, 1993]: the replication of subtrees required to represent some concepts and the rapid fragmentation of the training set data when attributes that can take a high number of values are tested at a node.

Decision graphs have been proposed as one way to alleviate these problems. The problem of selecting the minimum decision graph consistent with a training set is NP-complete [Takenaga and Yajima, 1993] and the need for heuristics is clear but the algorithms proposed to date for the construction of these graphs have met limited acceptability. One natural approach is to identify related subtrees in a decision tree obtained using standard methods but other authors have reported limited success using this method [Mahoney and Mooney, 1991]. Another approach uses a greedy algorithm that performs either a join or a split operation, depending on which one reduces more the description length [Oliver, 1993]. The author reported improvements over the use of decision trees in relatively simple problems but our experiments using a similar approach failed in more complex test cases because the algorithm tends to perform premature joins in complex problems. Very recently, a new approach was proposed [Kohavi, 1995] that reportedly represents an improvement over a previous algorithm [Kohavi, 1994] by the same author. This last approach is also based on the identification of common subtrees in a decision tree, but this tree is constrained to exhibit the same ordering of tests for all possible paths in the tree. A combination of this approach with some of the techniques introduced here may be of interest.

The approach described in this paper is radically different from any of these solutions. It uses well known induction algorithms [Quinlan, 1986, Pagallo and Haussler, 1990] and techniques for the manipulation of RODGs [Brace et al., 1989] in the generation of the initial decision graph. The algorithm then derives a compact decision graph by performing incremental changes until a graph that corresponds to a solution with minimal description length [Rissanen, 1978] is obtained. The problem of selecting an appropriate ordering for the variables is solved in a novel way using one of the highly effective heuristic algorithms for variable reordering proposed in the logic synthesis literature [Rudell, 1993].

Although it is possible to prove that, in general no bias is inherently superior to any other bias [Schaffer, 1994], we argue, by showing the results of a series of experiments, that the selection of small decision graphs is appropriate for many interesting problems.

2 DECISION TREES AND DECISION GRAPHS

We address the problem of inferring a classification rule from a labeled set of instances, the training set. In particular, we are interested in supervised single concept learning in discrete spaces. Let the input space be defined by a set of discretely valued attributes (or variables), $D = X_1 \times X_2 \ldots \times X_N$. Each instance $(d^i, t^i) \in D \times \{0, 1\}$ is described by a collection of discretely-valued attributes that defines a point in D

and a label that is 1 iff this point is a positive instance of the target concept. We assume that this labeling was obtained using an unknown (but fixed) function $\gamma(d) : D \rightarrow [0,1]$ that gives the probability that a given point in D is labeled with a 1. In the special case where $\gamma(d)$ is a function from D to $\{0,1\}$ the labeling is deterministic. The training set is a set of instances $T = \{(d^1, t^1), \ldots, (d^m, t^m)\}$. The objective is to infer a classification rule, or hypothesis, a function $f : D \rightarrow \{0,1\}$ that is used to predict the labels of unseen elements of D.

The values $t^1, \ldots, t^m$ define a Boolean vector t with m components that has a 1 in the ith position iff d^i is a positive instance. Boolean vectors will also be used in other contexts and will be manipulated using the standard Boolean operations in the natural way. The norm of a vector, $|z|$, represents the number of ones in z. When describing expressions that involve Boolean functions or vectors, we will follow the accepted conventions in the logic synthesis literature by omitting the conjunction operator and using the $+$ symbol to represent disjunction and the $\oplus$ symbol to represent the exclusive or operator.

We are interested in two representations for functions defined over the discrete domain D: decision trees and decision graphs. A decision tree is a rooted tree where each non-terminal node is labeled with the index of the attribute tested at that node. From each non-terminal node labeled with variable x_i, $|X_i|$ arcs labeled with the possible values of the attribute x_i point to the descendent nodes. In the case of single concept learning, there are two types of terminal (or leaf) nodes: type 0 and type 1, denoting, respectively, non-inclusion and inclusion in the target concept.

Similarly, a decision graph is a directed acyclic graph where each non-terminal node is labeled with the index of the attribute tested at that node. A decision graph is similar to a decision tree except that the underlying graph may have re-convergent paths. Both can be used to classify any instance by tracing a path from the root to a terminal node that follows the edges labeled with the values of the attributes present in that instance.

A decision graph is called *ordered* if there is an ordering of the variables such that, for all possible paths in the graph, the variables are always tested in that order (possibly skipping some of them). A decision graph is called *reduced* if no two nodes exist that branch exactly in the same way, and it is never the case that all outgoing edges of a given node terminate in the same node [Bryant, 1986]. A graph that is both reduced and ordered is called a *reduced ordered decision graph* (RODG).

In this work, only problems defined over Boolean spaces will be considered. Problems defined by multi-valued attributes are first mapped into a Boolean space by replacing each multi-valued attribute $x_i \in X_i$ by a set of $\lceil \log_2 |X_i| \rceil$ Boolean valued attributes. We will henceforth assume that all attributes are Boolean valued and that $D = \{0,1\}^N = B^N$. [Oliveira, 1994] discusses the advantages and drawbacks of this approach.

Consider now a decision tree or a decision graph defined over the domain D. Let $n_0, n_1, ..., n_s$ be the nodes in the tree or graph. Let v_i denote the variable tested in node n_i, n_{t_i} (the *then* node) denote the node pointed to by the arc leaving node n_i when v_i is 1 and n_{e_i} (the *else* node) denote the node pointed to by the arc leaving node n_i when v_i is 0. Finally let node n_s be the root of the decision tree or graph. Each node n_i defines a Boolean function $f(n_i) : D \rightarrow B$ defined, in a recursive way, by

$$f(n_i) = \begin{cases} f(n_{t_i}) & \text{if } v_i = 1 \\ f(n_{e_i}) & \text{if } v_i = 0 \end{cases} \tag{1}$$

The recursion stops when n_{e_i} (or n_{t_i}) is a terminal node. In this case $f(n_{e_i})$ (or $f(n_{t_i})$) equals the constant 0 function or the constant 1 function, depending on the type of the terminal node.

Finally, let y_i denote the Boolean vector with m components that has a 1 in position j iff the function defined by node n_i has the value 1 for the jth instance in the training set:

$$y_i^j = f_i(d^j) \tag{2}$$

3 MANIPULATING DISCRETE FUNCTIONS USING RODGS

Packages that manipulate reduced ordered decision graphs are widely available [Brace et al., 1989] and have become the most commonly used tool for discrete function manipulation in the logic synthesis community. A detailed exposition of the algorithms used to manipulate RODG structures is outside the scope of this paper and this section intends to introduce only the basic ideas. Several functions can be represented using a multi-rooted (or shared) RODG and each function is usually represented by a pointer to the RODG node that represents the function. The algorithms described in this paper make use of only a small fraction of the facilities provided by RODG packages, namely the following ones:

- Boolean combination of two existing functions. Example: $f := gh$.
- Complement of an existing function: $f := \overline{g}$.
- Creation of a function from an existing variable. For example, $f :=$ Fvar(i) returns a function f that is 1 when variable x_i is 1 and is 0 otherwise.
- The *if-then-else* operator. For example, $f :=$ Ite(v, g, h) returns $f := (vg) + (\overline{v}h)$.

Example 1 *The pseudo-code in figure 1 exemplifies how the function $f : \{0,1\}^4 \rightarrow \{0,1\}$ defined in expression (3) can be obtained using the primitives provided by the package. Figure 2 shows the successive RODGs created by the package to represent the functions g, h and f. In this and succeeding figures, the leftmost edge for each node represents the edge taken when the value of the attribute tested is 0.*

$$f(x_1, x_2, x_3, x_4) = \begin{cases} x_2 x_4 & \text{if } x_1 = 1 \\ \overline{x_3} + x_4 & \text{if } x_1 = 0 \end{cases} \tag{3}$$

```
Main()
    Bdd f, g, h               Declares f,g and h as functions
    InitializePackage()
    g := Fvar(2)Fvar(4)                      Computes x2x4
    h :=!Fvar(3) + Fvar(4)                   Computes !x3 + x4
    f := Ite(Fvar(1), g, h)
```

Figure 1: Computing the function f

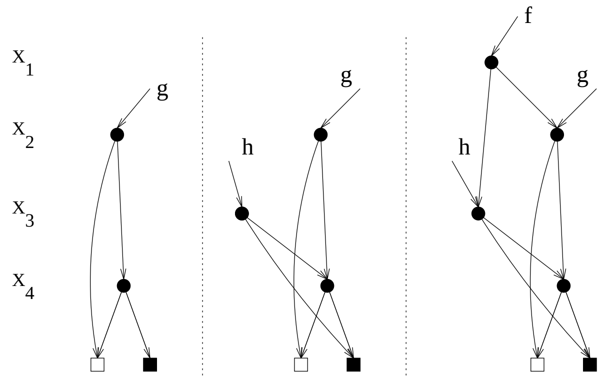

Figure 2: Successive RODGs created to represent f

The algorithm that manipulates RODGs is based on Shannon's decomposition

$$f = v f_v + \overline{v} f_{\overline{v}} \tag{4}$$

where v is a variable and f_v and $f_{\overline{v}}$ represent f evaluated at $v = 1$ and $v = 0$, respectively.

Now, let w be a variable, $f(n_i) = \text{Ite}(v_i, f(n_{t_i}), f(n_{e_i}))$ and assume that w comes before v_i in the ordering or that $v_i = w$. Finding the cofactors of f with respect to w is trivial because, in the first case, f is independent of w:

$$f_w = \begin{cases} f & \text{if } v_i \neq w \\ f(n_{t_i}) & \text{if } v_i = w \end{cases} \quad f_{\overline{w}} = \begin{cases} f & \text{if } v_i \neq w \\ f(n_{e_i}) & \text{if } v_i = w \end{cases} \tag{5}$$

The following recursive definition gives a simple algorithm for the computation of the function $z = \text{Ite}(f, g, h)$. Let v be the top variable of f, g, h. Then,

$$\begin{aligned} z &= v z_v + \overline{v} z_{\overline{v}} \\ &= v(fg + \overline{f}h)_v + \overline{v}(fg + \overline{f}h)_{\overline{v}} \\ &= v(f_v g_v + \overline{f}_v h_v) + \overline{v}(f_{\overline{v}} g_{\overline{v}} + \overline{f}_{\overline{v}} h_{\overline{v}}) \\ &= \text{Ite}(v, \text{Ite}(f_v, g_v, h_v), \text{Ite}(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \\ &= (v, \text{Ite}(f_v, g_v, h_v), \text{Ite}(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \end{aligned} \tag{6}$$

The recursion always terminates because f_v, g_v and h_v do not depend on v. The terminal cases are:

$$\text{Ite}(1, f, g) = \text{Ite}(0, g, f) = \text{Ite}(f, 1, 0) = f \tag{7}$$

4 MINIMUM MESSAGE LENGTH AND ENCODING OF RODGS

The tradeoff between hypothesis simplicity and accuracy in the training data is controlled using the minimum description length principle of Rissanen. This very general principle states, in a simplified way, that the hypothesis that minimizes the *sum of the length of the hypothesis description with the length of the data encoded using this hypothesis as a way to predict the labels* is the one more likely to be accurate in unseen instances.

Let L_g be the description length of an RODG and L_d be the length of the message required to describe the exceptions to this RODG in a given training set. According to the minimum description length principle the RODG that minimizes the total description length, $L_g + L_d$, will exhibit the best generalization accuracy. However the minimization of different linear combinations of L_g and L_d is also consistent with a Bayesian interpretation of the MDLP and may be chosen according to different beliefs about the concepts distribution. Therefore, the algorithms described in section 5 aim at obtaining a decision graph that minimizes

$$\mathcal{C} = \alpha L_g + L_d \tag{8}$$

The computation of L_g, the description length of the RODG, is performed by evaluating the length of a particular encoding. The reduced ordered decision graph and the exceptions are encoded using a scheme inspired in the one proposed for the induction of decision trees [Quinlan and Rivest, 1989], but modified to take into account the fact that a node can be visited more than once and restricted to consider only decision graphs with two terminal nodes. An RODG is encoded as a bit string, where the nodes are encoded as follows:

- A node that was never visited before is encoded starting with a bit at 1 followed by the encoding of the variable tested at that node, followed by the encoding of the node pointed to by the *else* edge, followed by the encoding of the node pointed to by the *then* edge.
- A node that was visited before is encoded starting with 0 followed by a reference to the node.

The first node to be described is the root of the graph, and the two terminal nodes are considered visited from the beginning and assigned references 0 and 1. We ignore the issues related with the use of non-integral numbers of bits and we make the description less redundant by noting that when one is deeper in the decision graph not all variables can be usefully tested. Furthermore, when a reference to an already described node is given, only $\log_2(r)$ bits are needed, where r is the number of nodes described up to that point.

We now need to compute L_d, the description length of the exceptions to a given RODG present in the training set. Exceptions to the RODG will be encoded as a string of 0's and 1's where the 1's indicate the locations of the exceptions. In general, the strings have many more 0's than 1's. Assume that there are k 1's in the string and the strings are of length m, known in advance.

We can encode the string by first sending the value of k, which requires $\log_2(m)$ bits and then describing which string with k 1's we are referring to. The total description length for the exceptions is therefore:

$$L_d = \log_2(m) + \log_2 \binom{m}{k} \tag{9}$$

Using Stirling's formula to approximate the second term in (9) we obtain

$$L_d = mH\left(\frac{k}{m}\right) + \frac{\log_2\left(\frac{m}{2\pi k(m-k)}\right)}{2} + \log_2(m) + O\left(\frac{1}{m}\right) \tag{10}$$

where $H(p)$ is the usual entropy function

$$H(p) = -p\log_2(p) - (1-p)\log_2(1-p) \tag{11}$$

5 DERIVING AN RODG OF MINIMAL COMPLEXITY

5.1 GENERATING THE INITIAL RODG

There are several possible ways to generate an RODG that can be used as the starting point for the local optimization algorithm. Experiments have shown that three of them are particularly effective. The RODG selected as the initial solution is the one amongst the following three that has minimal description length:

- The RODG that realizes the function implemented by a decision tree derived from the training set data using standard decision tree algorithms.
- The RODG that realizes the function implemented by a decision tree defined over a new set of variables obtained using constructive induction techniques.
- The RODG obtained by applying the restrict heuristic to the function obtained by listing all positive instances in the training set.

5.1.1 Initialization Using Decision Trees

One possible way to initialize the RODG is to obtain a decision tree from the data and to convert the function obtained by the decision tree to RODG form.

Since the tradeoff between hypothesis complexity and accuracy in the training set will be controlled by the local optimization algorithm, we used a simple decision tree approach [Quinlan, 1986] and performed no pruning on the tree derived using this method. Once the decision tree has been obtained, the reduced ordered decision graph is obtained applying expression (1) to the root of the decision tree and using the RODG package to manipulate the intermediate Boolean functions obtained.

Example 2 *Figure 3 shows one decision tree for the function $f = x_1x_2 + x_3x_4$ and the decision graph that results, assuming the ordering used is (x_1, x_2, x_3, x_4).*

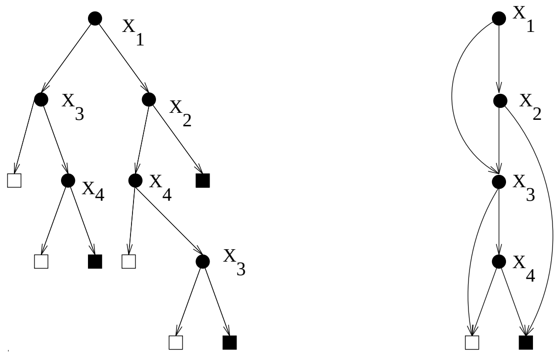

Figure 3: Decision tree for $f = x_1x_2 + x_3x_4$ and the corresponding decision graph

5.1.2 Initialization Using a Constructive Induction Algorithm

Constructive induction algorithms create new complex attributes by combining existing attributes in ways that make the description of the concept easier. The *fulfringe* constructive induction algorithm [Oliveira and Vincentelli, 1993] belongs to a family of constructive induction algorithms that identify patterns near the fringes of the decision tree and uses them to build new attributes [Pagallo and Haussler, 1990, Yang et al., 1991]. *Fulfringe* identifies the patterns shown in figure 4 which are a superset of the ones identified by *dcfringe*.

The new composite attributes are added (if they have not yet been generated) to the list of existing attributes and a new decision tree is built. The process

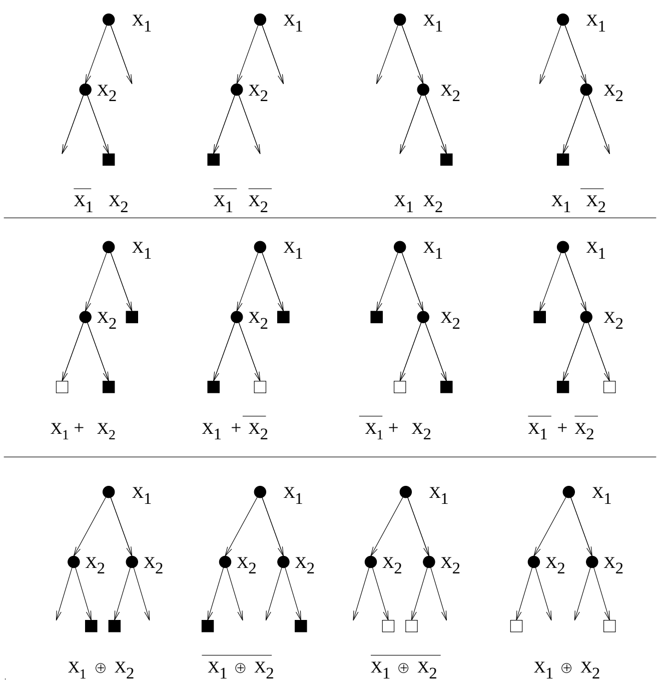

Figure 4: Fringe patterns identified by *fulfringe*

is iterated until no further reduction in the decision tree size takes place or a decision tree with only one decision node is built.

Since the composite attributes are Boolean combination of existing ones the RODGs for them are created in a straightforward way using the Boolean operations between existing functions provided by the RODG package. Expression (1) can still be used to derive the RODG implemented by a decision tree defined over this extended set of variables but the variable v_i will not refer, in general, to a primitive attribute. This is handled by the functions available in the RODG package in a transparent way.

Note that even though the successive decision trees are defined using composite attributes, the RODGs that correspond to any one of these trees are still defined over the original set of variables. In this way, the constructive induction algorithm is used only to derive a simpler Boolean function to initialize the RODG reduction algorithm, not to add new variables as in standard constructive induction approaches.

Example 3 *Figure 5 shows the successive decision trees obtained using this algorithm for the function used in example 2. The first decision tree created is the same as before. Using the patterns listed in figure 4 the algorithm creates the two following attributes:* $x_5 = x_1 x_2$ *and* $x_6 = x_3 x_4$. *RODGs for these new attributes are also created, as they will be needed in the next phase.*

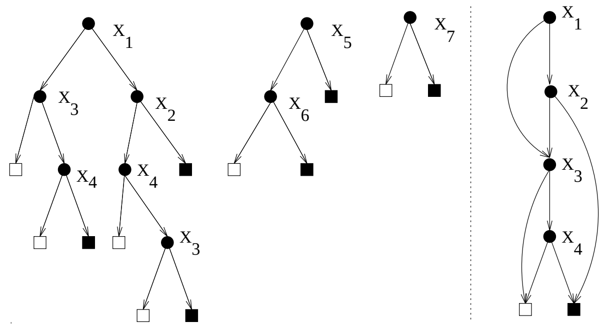

Figure 5: Decision trees created by *fulfringe*

A smaller decision tree is then built using these attributes (together with the primitive ones, in general) and the new attribute $x_7 = x_5 + x_6$ *is created, as well as the RODG for* x_7 *as a function of* $(x_1 \ldots x_4)$.

The final decision tree has a single node that tests attribute x_7. *The RODG is created by applying expression (1). In this case, the final RODG is the same as the one obtained using the initial decision tree although this is not always the case.*

5.1.3 Initialization Using the Restrict Operator

The third way to initialize the algorithm is to use the restrict operator [Coudert et al., 1989]. This RODG operator can be used to obtain a more compact RODG representation for an incompletely specified function.

The restrict operator belongs to a family of heuristics that generate a small RODG by merging, in a bottom up fashion, nodes in an RODG. The merging of nodes is performed in a way that keeps the RODG consistent with the training set data. Two RODGs are required to apply the restrict heuristic: an RODG that describes the function f to be *restricted* and an RODG that describes the *care set*, i.e., the points in the input space where the value of the function is relevant and should not be changed. The first RODG is created by considering a function that is 1 for all positive instances and 0 otherwise. The care set consists off all the points in the input space that are present in the training set, either as positive or negative instances of the target concept[1]. The restrict heuristic is then applied to obtain a small RODG that is consistent with the training set but smaller than the original one.

The restrict heuristic is remarkably fast and obtains, in some cases, RODGs that are much better solutions

[1]If there exists conflicting information in the training set, i.e., instances with the same values of the attributes but different labels, these points are also considered as belonging to the don't care set. This is a sub-optimal solution and this heuristic does not work well in training sets that exhibit high levels of noise.

than the ones obtained by the much slower decision tree algorithms. However, in problems that have many attributes and where the positive and negative instances can be separated by a small subset of the available attributes this heuristic tends to generate RODGs that depend only on a small subset of the attributes, usually the ones that come first in the ordering selected.

5.2 REDUCING AN RODG

The search for an RODG of minimum total description length is performed in steps. Each step decreases the description length of the RODG but may also cause the creation of new exceptions and is accepted only if the total description length decreases.

At each step, one or more nodes are removed from the RODG. To each node n_i in the RODG is associated a Boolean vector w_i that contains a 1 for the positions j (i.e. w_i^j) that correspond to instances in the training set that define paths in the RODG that go through node n_i and contains 0 otherwise. The family of vectors w_i can be computed by applying recursively the following Boolean expressions:

$$w_s^j = 1 \tag{12}$$

$$w_i = \Big(\sum_{j:n_{e_j}=n_i} \overline{v_j} w_j \Big) + \Big(\sum_{j:n_{t_j}=n_i} v_j w_j \Big) \tag{13}$$

5.2.1 Removing One Node by Redirecting Incoming Edges

The *RemoveNode* procedure reduces the description length by making one of the nodes in the RODG redundant. This is done by redirecting all its incoming edges. When node n_i is under consideration, the algorithm goes through all incoming edges and selects, for each one of them, a different node, as shown in figure 6.

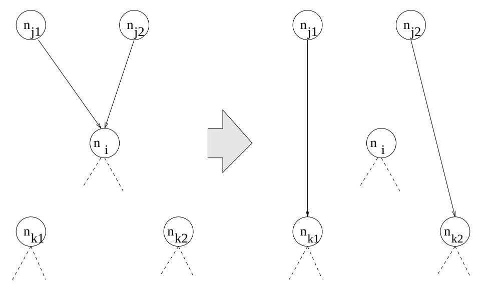

Figure 6: Removing one node from the RODG

The value of the function implemented by this node is relevant only for the instances that reach n_i through the edge that is being redirected. The node that, for these instances, implements a function more similar to n_i is selected as the new destination for the edges coming into node n_i. The pseudo-code in figure 7 describes how this modification is accomplished.

RemoveNode(R)
    foreach n_i
        foreach j s.t. $n_{e_j} = n_i$ *Left children of* n_i
            $w := w_j \overline{v_j}$ *Instances using this edge*
            Select n_k such that $|(y_k \oplus t)w|$ is minimal
            Modify RODG such that $n_{e_j} = n_k$
        foreach j s.t. $n_{t_j} = n_i$ *Right children of* n_i
            $w := w_j v_j$ *Instances using this edge*
            Select n_k such that $|(y_k \oplus t)w|$ is minimal
            Modify RODG such that $n_{t_j} = n_k$
        if New RODG has smaller description length
            return (New RODG)
        else
            Undo changes
    return (Failure)

Figure 7: The *RemoveNode* procedure.

5.2.2 Replacing Two Nodes by a New Node

If the procedure *RemoveNode* fails to make one node redundant, the more expensive procedure *ReplacePair* is called. *ReplacePair* removes a pair of nodes by creating a new node that implements a function as close as possible to the functions implemented by the pair of nodes under consideration (see figure 8). The value of the new function is only relevant for the instances that reach the nodes being considered for removal as shown in the pseudo-code in figure 9.

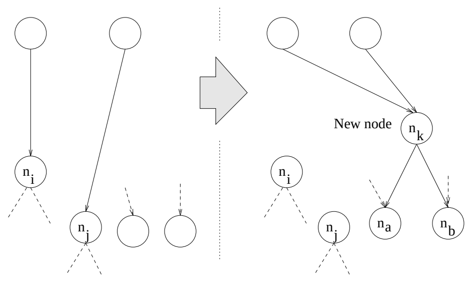

Figure 8: Replacing a pair of nodes by a new node.

5.2.3 Complexity Considerations

The complexity of these two procedures dominates the run time of the algorithm. Because the effect of each change can be estimated locally the re-computation of the overall description length of the RODG or the number of exceptions created by a local modification is not required. With careful coding, the *RemoveNode* procedure requires $O(s^2 m)$ operations, where s is the number of nodes in the current RODG and m is the

```
ReplacePair(R)
    foreach n_i
        foreach n_j                      For each pair of nodes
            w := w_i + w_j     Instances reaching n_i or n_j
            Select v_k, n_a and n_b s.t.
                |(Ite(v_k, n_b, n_a) ⊕ t)w| is minimal.
            Create n_k = Ite(v_k, n_b, n_a)
            Remove n_i and n_j and redirect edges to n_k
            if New RODG has smaller desc. length
                return (New RODG)
            else
                Undo changes
    return (Failure)
```

Figure 9: The *ReplacePair* procedure

size of the training set. The *ReplacePair* procedure is more expensive and requires $O(s^3m)$ operations. By using bit packing techniques the algorithm can be applied to reduce RODGs with hundreds or a few thousands of nodes.

5.3 SELECTING THE BEST ORDERING

The selection of a good ordering for the variables is important if the target is to obtain a compact RODG. Selecting the optimal ordering for a given function is NP-complete [S. Tani and Yajima, 1993] but many heuristic algorithms have been proposed for this problem. Our implementation uses the *sift* algorithm for dynamic RODG ordering [Rudell, 1993]. This approach is based on the fact that swapping the order of two variables in the RODG ordering can be done very efficiently because only the nodes in these two levels are affected. The *sift* algorithm selects the best position in the ordering for a given variable by moving that variable up and down (using the inexpensive swap operation) and recording the smaller size observed. This procedure is applied once to all variables and can be, optionally, iterated to convergence.

6 EXPERIMENTS

6.1 RESULTS IN PROBLEMS FROM THE LITERATURE

In this section, the comparison between different algorithms was made using a set of problems that have been addressed in the machine learning literature [Pagallo and Haussler, 1990, Murphy and Aha, 1991] and are described in detail elsewhere [Oliveira, 1994].

6.1.1 Experimental Setup

The preferred approach to perform the performance comparison between different algorithms is to use multiple runs with fixed training set sizes. This approach raises the problem of selecting the training set size in such a way that the problem is non-trivial but that enough data is present to allow for meaningful induction. One solution is to select the size of the training set size as a function of the complexity of the target concept when expressed using a given representation scheme but this approach is difficult to use when two very different representation schemes are used.

Instead of evaluating the performance of the algorithms using only training sets of one fixed size, we computed the average performance for training sets of increasing size. For each problem, we started by selecting a data set. This test set contains either all the available data (for the problems for which a limited amount of data is available), or a set of 5000 instances randomly generated and labeled in accordance with the target concept. We then selected 10 different sizes for the training sets, up to a thousand instances (or the size of the data set, if smaller) randomly generated in such a way that the larger training sets always include the smaller ones. Each *experiment* consists in evaluating the performance of the programs using these 10 training sets. The *experiment error* for each of the programs is obtained by averaging the generalization error in the instances in the dataset not used for training over the 10 different training set sizes. The *experiment difference* is obtained by subtracting the experiment errors of the two algorithms under comparison.

For each problem we performed 10 experiments and computed the *average error* by averaging the 10 *experiment errors*. For the 10 *experiment differences* obtained, we computed not only the average value (the *average difference*) but also the observed variance of this value.

In order to evaluate the statistical significance of the observed differences, we used the fact that, if μ_z, $\overline{z}$ and $\hat{\sigma}_z^2$ are respectively the real mean, the observed mean and the observed standard deviation of the differences in n experiments then

$$t = \frac{(\overline{z} - \mu_z)\sqrt{n}}{\hat{\sigma}_z} \tag{14}$$

has approximately a Student's t distribution with $n-1$ degrees of freedom. All significance tests were performed at a significance level of 0.05.

6.1.2 Comparison with C4.5

Table 1 lists the *average error* for the `C4.5` and `smog` algorithms in this set of problems as well as the *average difference*. A positive difference means that `C4.5` showed a larger error than `smog` in a particular problem. Both `C4.5` and `smog` were run using the default parameters (The default value for α, in expression (8) is 0.5). A circle in a given row marks the algorithm that obtained the lowest average error in the given

Problem	Average error				Difference
		C4.5		smog	$\overline{z} \pm \sigma_z$
dnf1		23.04	○	22.35	0.69 ± 1.29
dnf2		20.00	●	11.62	8.37 ± 1.39
dnf3		13.67	●	8.72	4.95 ± 1.16
dnf4		35.86	●	21.81	14.05 ± 2.34
mux6		1.41	●	0.19	1.22 ± 0.91
mux11		19.57	●	5.12	14.45 ± 1.76
par4_16		37.43	●	0.79	36.64 ± 6.26
par5_32		49.18	●	21.91	27.28 ± 4.91
monk1		6.92	●	0.62	6.30 ± 1.56
monk2		34.89	●	17.25	17.63 ± 2.57
monk3		1.09	●	0.50	0.58 ± 0.65
vote	●	4.32		6.04	-1.72 ± 0.72
mushroom		1.72	○	1.47	0.26 ± 0.70
splice	○	11.55		11.69	-0.14 ± 0.92
tictactoe		9.25	●	6.30	2.95 ± 0.82
breast	●	4.52		6.41	-1.89 ± 1.00
kkp		11.23	●	1.53	9.70 ± 2.06
krkp		3.09		3.09	0 ± 0.33
heel9		7.96	●	1.62	6.34 ± 0.85
heel		25.83	●	2.58	23.25 ± 1.36
sm12		10.51	●	2.06	8.45 ± 0.71
str18		11.76	●	6.31	5.46 ± 1.24

Table 1: Average errors for C4.5 and smog

problem. A filled circle means that the difference observed is statistically significant.

6.2 RESULTS IN THE WRIGHT LABORATORY BENCHMARK SET

The results obtained in this set of problems were obtained by Timothy Ross and the Pattern Theory Group at the Air Force Wright Laboratory. They kindly agreed to run smog in a benchmark his group assembled [Goldman, 1994] for the purpose of evaluating the effectiveness of a set of learning algorithms.

Each one of the problems is defined over a space of 8 Boolean attributes. The setup is similar to the one used in section 6.1 but the the authors of this benchmark evaluate the generalization accuracy using the whole dataset. The test set therefore includes the training set data. One experiment consists of 10 independent runs with increasing training set sizes. The training set sizes selected are the multiples of 25 between 25 and 250 and 10 experiments were performed for each problem.

Table 2 shows the results obtained for the problems in this set. The meaning of the column labels is the same as before. The results listed for the program C4.5 were obtained with the options found to give the best overall results for this program. Smog was run with α (in expression (8)) set to a low value because the problems are known to be noise free. The generalization error obtained with smog was smaller in a statistical significant way for a large number of problems.

Problem	Average error				Difference
		C4.5		smog	$\overline{z} \pm \sigma_z$
add0		6.09	●	4.73	1.35 ± 0.80
add2		10.49	●	8.02	2.46 ± 2.12
add4		1.68	●	0.25	1.43 ± 1.67
and_or_chain8		1.50	○	1.40	0.10 ± 0.35
ch15f0		11.34	●	7.96	3.38 ± 1.48
ch176f0		1.50	○	0.94	0.56 ± 1.78
ch177f0		0.94	●	0.00	0.94 ± 1.33
ch22f0		5.15	○	4.71	0.44 ± 1.00
ch30f0		4.70	○	4.68	0.02 ± 0.89
ch47f0		7.43	●	6.48	0.94 ± 1.34
ch52f4		8.52	●	7.86	0.65 ± 0.87
ch70f3		3.94	●	3.48	0.46 ± 0.50
ch74f1	○	5.52		5.53	-0.01 ± 0.78
ch83f2		9.79	○	9.76	0.03 ± 0.73
ch8f0		5.41	●	4.83	0.58 ± 0.84
contains_4_ones		21.95	●	13.99	7.96 ± 0.79
greater_than		7.09	●	5.79	1.31 ± 0.83
interval1		13.46	●	11.18	2.29 ± 0.82
interval2		17.55	●	14.04	3.51 ± 0.81
kdd1		0.38	○	0.13	0.25 ± 0.79
kdd2		1.61	○	1.38	0.23 ± 0.42
kdd3		0.81		0.81	0.00 ± 0.00
kdd4		0.00		0.00	0.00 ± 0.00
kdd5		4.05	●	3.31	0.74 ± 0.47
kdd6	○	0.97		1.13	-0.16 ± 0.47
kdd7		7.84	●	2.43	5.41 ± 0.78
kdd8		1.92	○	1.75	0.17 ± 0.92
kdd9		4.64	●	3.59	1.05 ± 1.07
kdd10		7.52	●	4.77	2.75 ± 1.06
majority_gate		13.82	●	9.57	4.25 ± 0.66
modulus2		4.78	○	4.64	0.14 ± 0.28
monkish1		2.38	●	1.25	1.12 ± 0.87
monkish2		17.43	●	12.85	4.57 ± 1.12
monkish3		1.84	○	1.72	0.12 ± 0.34
mux8		5.47	●	3.91	1.56 ± 1.41
nnr1		17.50	●	11.80	5.70 ± 1.07
nnr2		2.25	○	1.69	0.56 ± 1.19
nnr3		11.63	●	10.96	0.66 ± 0.77
or_and_chain8	○	1.07		1.19	-0.12 ± 0.32
pal		6.34	●	5.21	1.13 ± 1.11
pal_dbl_output		19.95	●	18.13	1.82 ± 0.93
pal_output		23.04	○	23.00	0.04 ± 0.50
parity		33.28	●	14.88	18.40 ± 3.26
primes8		13.39	○	13.35	0.04 ± 0.89
remainder2	○	10.88		11.01	-0.13 ± 0.85
rnd1		24.25	●	23.75	0.50 ± 0.48
rnd2	○	23.99		24.02	-0.04 ± 0.61
rnd3	○	23.25		23.43	-0.18 ± 0.55
rnd_m1	●	0.84		0.96	-0.11 ± 0.10
rnd_m5	●	3.05		3.54	-0.48 ± 0.42
rnd_m10	●	4.99		5.49	-0.50 ± 0.69
rnd_m25		10.05	○	9.88	0.18 ± 0.63
rnd_m50		16.95	○	16.58	0.37 ± 0.84
rndvv36		15.07	●	13.05	2.01 ± 1.17
substr1		10.96	●	6.76	4.20 ± 1.06
substr2		8.75	●	6.66	2.09 ± 1.10
subtraction1		15.80	●	13.59	2.21 ± 1.07
subtraction3		1.68	●	0.25	1.43 ± 1.67

Table 2: Errors for the Wright Labs benchmark

7 CONCLUSIONS AND FUTURE WORK

We presented an algorithm for the induction of reduced ordered decision graphs from data. The experimental results have shown that for many problems defined over discrete spaces, the bias for small RODGs is appropriate and the generalization accuracy is better than the one obtained by alternative techniques.

The algorithms described in this paper are considerably slower than standard decision tree algorithms. This slowdown may be acceptable in many applications but be intolerable in others. Ultimately, the user has to decide if the increased generalization accuracy is important enough to offset the extra CPU time. We believe this will be the case in many applications.

In the present version, smog can be used as a direct replacement for popular decision tree algorithms like C4.5 for problems that are defined in a natural way using discretely valued attributes. A pre-processing step may be used to convert continuous valued variables into discrete ones, but this is currently done in a sub-optimal way and the advantages of the decision graph approach may be partially lost. This limitation makes it clear that the most interesting direction for future research is the definition of a procedure to select tests on continuously valued attributes that takes into account the specific characteristics of decision graphs.

Another direction for future work is related with the fact that the current version of smog manipulates the representation internally using binary RODGs. For some problems, it may be advantageous to use multi-valued RODGs directly.

References

[Brace et al., 1989] Brace, K., Rudell, R., and Bryant, R. (1989). Efficient implementation of a BDD package. In *Design Automation Conference.*

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees.* Wadsworth International Group.

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers.*

[Coudert et al., 1989] Coudert, O., Berthet, C., and Madre, J. C. (1989). Verification of synchronous sequential machines based on symbolic execution. In Sifakis, J., editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag.

[Goldman, 1994] Goldman, J. A. (1994). Machine learning: A comparative study of pattern theory and C4.5. Technical Report WL-TR-94-1102, Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543.

[Kohavi, 1994] Kohavi, R. (1994). Bottom-up induction of oblivious read-once decision graphs. In *European Conference in Machine Learning.*

[Kohavi, 1995] Kohavi, R. (1995). Oblivious decision trees, graphs and top-down pruning. In *IJCAI-95.* to be published.

[Mahoney and Mooney, 1991] Mahoney, J. J. and Mooney, R. J. (1991). Initializing ID5R with a domain theory: some negative results. Technical Report 91-154, CS Dept., University of Texas at Austin, Austin, TX.

[Murphy and Aha, 1991] Murphy, P. M. and Aha, D. W. (1991). *Repository of Machine Learning Databases - Machine readable data repository.* University of California, Irvine.

[Oliveira, 1994] Oliveira, A. L. (1994). *Inductive Learning by Selection of Minimal Complexity Representations.* PhD thesis, UC Berkeley. Also available as UCB/ERL Technical Report M94/97.

[Oliveira and Vincentelli, 1993] Oliveira, A. L. and Vincentelli, A. S. (1993). Learning complex boolean functions : Algorithms and applications. In *Advances in Neural Information Processing Systems 6*, Denver, CO. Morgan Kaufmann.

[Oliver, 1993] Oliver, J. J. (1993). Decision graphs - an extension of decision trees. Technical Report 92/173, Monash University, Clayton, Victoria 3168, Australia.

[Pagallo and Haussler, 1990] Pagallo, G. and Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–100.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

[Quinlan and Rivest, 1989] Quinlan, J. R. and Rivest, R. L. (1989). Inferring decision trees using the Minimum Description Length Principle. *Inform. Comput.*, 80(3):227–248.

[Rissanen, 1978] Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14:465–471.

[Rudell, 1993] Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, pages 42–47. IEEE Computer Society Press.

[S. Tani and Yajima, 1993] S. Tani, K. H. and Yajima, S. (1993). The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Algorithms and Computation, 4th Internation Symposium*, pages 389–98.

[Schaffer, 1994] Schaffer, C. (1994). A conservation law for generalization performance. In *Proceedings of the Eleventh International Conference in Machine Learning*, San Mateo. Morgan Kaufmann.

[Takenaga and Yajima, 1993] Takenaga, Y. and Yajima, S. (1993). NP-completeness of minimum binary decision diagram identification. Technical Report COMP 92-99, Institute of Electronics, Information and Communication Engineers (of Japan).

[Yang et al., 1991] Yang, D. S., Rendell, L., and Blix, G. (1991). Fringe-like feature construction: A comparative study and a unifying scheme. In *Proceedings of the Eight International Conference in Machine Learning*, pages 223–227, San Mateo. Morgan Kaufmann.