Accepted to ECML-94

Bottom-Up Induction of Oblivious Read-Once Decision Graphs

Ron Kohavi

Computer Science Department, Stanford University
Stanford, CA. 94305
E-mail: Ronnyk@CS.Stanford.EDU

Abstract. We investigate the use of *oblivious, read-once decision graphs* as structures for representing concepts over discrete domains, and present a bottom-up, hill-climbing algorithm for inferring these structures from labelled instances. The algorithm is robust with respect to irrelevant attributes, and experimental results show that it performs well on problems considered difficult for symbolic induction methods, such as the Monk's problems and parity.

1 Introduction

Top down induction of decision trees [25, 24, 20] has been one of the principal induction methods for symbolic, supervised learning. The tree structure, which is used for representing the hypothesized target concept, suffers from some well-known problems, most notably the replication problem and the fragmentation problem [23]. The replication problem forces duplication of subtrees in disjunctive concepts, such as $(A \wedge B) \vee (C \wedge D)$; the fragmentation problem causes partitioning of the data into fragments, when a high-arity attribute is tested at a node. Both problems reduce the number of instances at lower nodes in the tree — instances greatly needed for statistical significance of tests performed during the tree construction process.

The smallest decision trees for most symmetric functions, such as parity and majority, have exponential size; consequently, programs that look for small trees tend to generalize poorly on such functions. One notable advantage of decision trees over other representations, such as neural nets, is the fact that the learned structures are readable by human experts who can confirm, reject, or modify the given hypothesis (*i.e.*, the tree), aided by background knowledge.

In this paper we investigate the use of *Oblivious, read-Once Decision Graphs* (**OODGs**) as a structure for representing concepts over discrete domains, and present a bottom-up, hill-climbing algorithm for inferring these structures from instances. OODGs have a *different bias* from that of decision trees, and thus some concepts that are hard to represent as trees are easy to represent as OODGs and vice-versa (although the latter seems rare). Since OODGs are graphs, they are easy for humans to perceive, and should be preferred over other representations (*e.g.*, neural nets) whenever it is important to comprehend the meaning and structure of the induced concept.

OODGs (see Sect.2 for a formal definition) are rooted, directed acyclic graphs (DAGs) that can be divided into levels. All nodes at a level test the same attribute, and all the edges that originate from one level terminate at the next level. See Fig.1(a) for an example of an OODG representing the exclusive-or (denoted by $\oplus$) of three variables, X_1, X_2, and X_4, with a fourth variable, X_3, being irrelevant.

The restriction that all nodes of a given level must test the same attribute gives the structure many nice properties (see Sect.2.1), and has proven to be very useful in the engineering community, where it is used in Ordered Binary Decision Diagrams (OBDDs). This bias, while constraining the class of decision graphs, still allows the "compression" of circuits with up to 10^{120} states into manageable structures, which are then used in automatic verification of correctness.

In the next section we formally define the OODG structure and present some basic properties. Section 3 introduces the basic algorithm, which is nondeterministic and assumes that the full instance space is available. Section 4 presents a hill-climbing version of the algorithm using one level of lookahead to resolve both the nondeterminism and the problem of generalization from an incomplete instance space. An important observation is the ability of the algorithm to discover irrelevant attributes, helping to decrease the size of the hypothesis space, and making the generalization much easier. Section 5 reports preliminary experiments. Section 6 describes related work, and Section 7 concludes with future work.

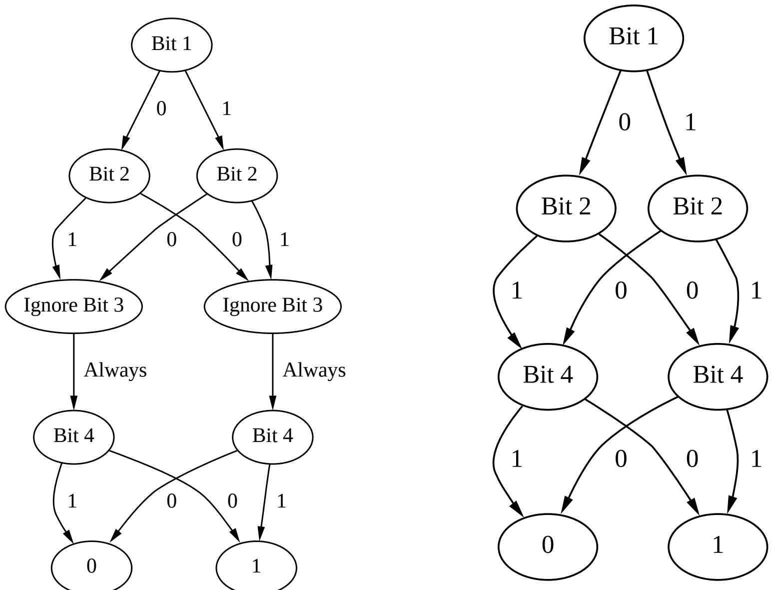
2 Oblivious Read-Once Decision Graphs

In this section we formally define the OODG structure and describe some basic properties. The name OODG is a combination of the terms "Oblivious" and "read-Once" that are used in theoretical complexity analysis of branching programs, and the term "Decision Graph" that is used in the artificial intelligence community, most notably the recent use of the term by Oliver, Dowe, and Wallace in [22, 21].

Given n discrete variables (or attributes), $X_1, X_2, \ldots, X_n$, with domains $D_1, \ldots, D_n$ respectively, the **instance space** $\mathcal{X}$ is the cross-product of the domains, *i.e.*, $D_1 \times \cdots \times D_n$. A ***k*-categorization function** is a function f mapping each instance in the instance space to one of k categories, *i.e.*, $f : \mathcal{X} \mapsto \{0, \ldots, k-1\}$. Without loss of generality, we assume that for each category there is at least one instance in $\mathcal{X}$ that maps to it.

2.1 The OODG Structure

We begin by describing a general decision graph, then specialize it to be read-once and oblivious.



(a) An OODG for $X_1 \oplus X_2 \oplus X_4$

(b) An OODG for $X_1 \oplus X_2 \oplus X_4$ with "constant" branching nodes removed.

Fig. 1. An OODG for 3-bit parity with one irrelevant attribute ($\oplus$ denotes exclusive-or).

A **decision graph** for a k-categorization function over variables $X_1, X_2, \ldots,$ X_n with domains $D_1, D_2, \ldots, D_n$, is a directed acyclic graph (DAG) with the following properties:

1. There are exactly k nodes, called **category nodes**, that are labelled $0, 1, \ldots,$ $k-1$, and have outdegree zero.
2. Non-category nodes are called **branching nodes.** Each branching node is labelled by some variable X_i and has $|D_i|$ outgoing edges, each labelled by a distinct value from D_i.
3. There is one distinguished node — the **root** — that is the only node with indegree zero.

The category assigned by a decision graph to a given variable assignment (an instance), is determined by tracing the (unique) path from the root to a category node, branching according to the labels on the edges.

A **read-once** decision graph is a graph where each variable occurs at most once along any computation path. A **levelled** decision graph is a graph where the nodes are partitioned into a sequence of pairwise disjoint sets, the levels, such that outgoing edges from each level terminate at the next level. An **oblivious** decision graph is a levelled graph such that all nodes at a given level are labelled by the same variable. An oblivious graph thus defines a total ordering on the variables. (The name "oblivious" denotes the fact that testing of variables depends only on their order within the levels, independent of the input itself.) An oblivious decision graph is **reduced** if there do not exist two distinct nodes at the same level that branch in exactly the same way on the same values. If two such nodes exist, they can be united.

An **OODG** is an oblivious read-once decision graph. Unless otherwise noted, the OODG is assumed to be reduced. The **size** of an OODG is the number of nodes in the graph, and the **width** of a level is the number of nodes at that level.

When displaying OODGs, if all edges emanating from a node terminate at the same node, we either replace the edges by one edge labelled "always," as shown in Fig.1(a), or remove such "constant" nodes altogether, as shown in Fig.1(b).

2.2 Properties of OODGs

We now describe some properties of OODGs. These properties help us understand the strengths and weaknesses of the OODG structure. Proofs of these properties for OBDDs can be found in [7, 6, 17], and can be generalized to OODGs.

- An OODG can represent any k-categorization function.
- For any k-categorization function f, and for a given ordering of the variables for the levels, there is a unique (up to isomorphism) reduced OODG implementing f.
  This is not surprising, as an OODG is very similar to a deterministic finite automaton that has been "unrolled" to avoid cycles.
- There exist functions that have a polynomial (or even linear) size OODG representation under one variable ordering, and an exponential size OODG under another ordering. One such example for Boolean variables is
  $$(X_1 \wedge X_2) \vee (X_3 \wedge X_4) \vee \cdots \vee (X_{2n-1} \wedge X_{2n}).$$
  The ordering $X_1, X_2, \ldots, X_{2n}$ gives a graph with $O(n)$ nodes, while the ordering $X_1, X_{n+1},\ X_2, X_{n+2},\ \ldots,\ X_n, X_{2n}$ requires $O(2^n)$ nodes.
- There are functions for which no variable ordering results in a polynomial size OODG representation (the Shannon effect). Wegener [28] has shown that almost all Boolean functions result in exponentially sized branching programs (and hence OODGs) under all orderings. Bryant [6] showed that at least one of the $2n$ bits of integer multiplication is an inherently complex function, requiring exponential sized OBDD (and hence OODG) for all orderings.
- All symmetric Boolean functions — functions which yield the same value for all permutations of the input variables — have OODGs of size $O(n^2)$. Examples of symmetric Boolean functions are parity, "exactly k-of-n", and "at least k-of-n".

We now present a theorem that gives an upper bound on the width of different levels of an OODG, given an instance space of Boolean variables. This theorem shows that the width bounds on OODGs are asymmetric; the graph grows much faster from the bottom than from the top. In Fig.2, the width of the "kite" at each level is proportional to the maximum number of nodes possible at that level. The kite is thus an envelope bounding both the overall size and the specific width of every OODG with the given number of levels. This theorem is one of the motivations for growing the graphs bottom-up — a wrong ordering will "explode" fast, and a lookahead of $\log n$ levels, will get us to the level where the graph can be the widest.

Theorem 1 Kite Theorem. *The width of level i of a reduced OODG with m levels implementing a k-categorization function over Boolean inputs is bounded by*

$$\min\left\{2^i, k^{2^{(m-i)}}\right\}$$

Proof (sketch): The first term is a top-down bound; each Boolean variable can cause the number of branching nodes to grow by a factor of at most two. The second term is a bottom-up bound; if a level has k nodes, the level above it must have at most k^2 nodes, since there are only k^2 mappings from $\{0,1\} \mapsto \{0,\ldots,k-1\}$. If two nodes branch the same way on both values of the Boolean variable, and the OODG is not reduced. □

3 Bottom-Up Construction of OODGs

In this section we present an algorithm for constructing a reduced OODG given the full (labelled) instance space. The algorithm is recursive and nondeterministic. For simplicity of notation, we assume Boolean variables and an arbitrary number of categories. Our current implementation allows general discrete variables.

The input to the algorithm is a set of sets, $\{C_0, C_1, \ldots, C_{k-1}\}$, where each set C_i is the set of all instances labelled with category i. The output of the algorithm is a reduced OODG that correctly categorizes the instance space.

The algorithm, shown in Fig.3, works by creating sets of instances, such that each set corresponds to one node in the graph (the input sets corresponding to the category nodes). Intuitively, we want an instance in a set C_i to reach node V_i corresponding to it, when the instance's path is traced from the root of the completed OODG, branching at branching-nodes according to the attribute values.

Given the input, the algorithm nondeterministically selects a variable X to test at the penultimate level of the OODG. It then creates new sets of instances (corresponding to the nodes in the penultimate level of the final OODG) which are projections of the original instances with variable X deleted. The sets are created in such a way that all instances in a set C_{xy} (which matches a branching node) are a subset of the instances of some input set C_x when augmented with

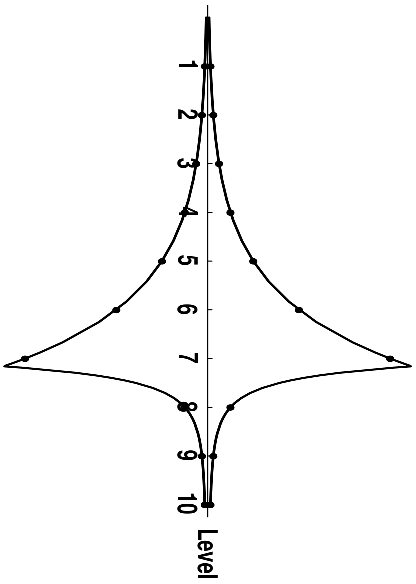


Fig. 2. The diagram's width at a level is proportional to the maximum number of nodes possible in an OODG at that level.

variable $X = 0$, and similarly, they are a subset of the instances of some set C_y when augmented with variable $X = 1$. In the graph, the branching node corresponding to C_{xy} will have the edge labelled 0 terminating at node x, and the edge labelled 1 terminating at node y.

The new sets now form a smaller problem over $n - 1$ variables, and the algorithm calls itself recursively to compute the rest of the OODG with the sets of the new level serving as the input. The recursion stops when the input to the algorithm is a single set, possibly consisting of the *null* instance (0 variables).

It can be shown that the algorithm always terminates with a reduced OODG (proof omitted).

Example 1. Executing the Algorithm on Parity In this example we show how to run the algorithm for the 3-bit odd parity function with one irrelevant attribute, *i.e.*, $f = X_1 \oplus X_2 \oplus X_4$ (X_3 being irrelevant). To resolve the nondeterminism, we will select attributes in reverse numerical order, that is, X_4, X_3, X_2, X_1.

The input to the algorithm is $\{C_0, C_1\}$. All instances in C_0 have a label 0, and all elements in C_1 have label 1

$$C_0 = \{0000, 0010, 0101, 0111, 1001, 1011, 1100, 1110\}$$
$$C_1 = \{0001, 0011, 0100, 0110, 1000, 1010, 1101, 1111\}$$

Deleting attribute X_4 from each instances gives us the following projected in-

Input: k sets $C_0, \ldots, C_{k-1}$ such that $\mathcal{X} = \bigcup_{i=0}^{k-1} C_i$ (the whole instance space).
Output: Reduced OODG correctly categorizing all instances in $\mathcal{X}$.

1. If $(k = 1)$ then return the graph with one node.
2. Nondeterministically select a variable X to be deleted from the instances.
3. Project the instances in $C_0, \ldots, C_{k-1}$ onto the instance space $\mathcal{X}'$, such that variable X is deleted. Formally, if X is the ith variable,
$$\mathcal{X}' \leftarrow \pi_{(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)} \bigcup_{i=0}^{k-1} C_i$$
4. For all $i, j \in \{0, \ldots, k-1\}$ let C_{ij} be the set containing instances from $\mathcal{X}'$ s.t. the instances belong to set C_i when augmented with $X = 0$, and to the set C_j when augmented with $X = 1$.
5. Let k' be the number of non-empty sets from $\{C_{ij}\}$. Call the algorithm recursively with the k' non-empty sets, and let G be the OODG returned.
6. Label the k' leaf nodes of G, corresponding to the non-empty sets C_{ij} with the variable X. Create a new level of k nodes corresponding to the sets $C_0, \ldots, C_{k-1}$. From the node corresponding to each C_{ij}, create two edges: one labelled 0, terminating at the (category) node corresponding to C_i, and the other labelled 1, terminating at the (category) node corresponding to C_j.
7. Return the augmented OODG G.

Fig. 3. A nondeterministic algorithm for learning OODGs.

stance space
$$\mathcal{X}' = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Because we started with the full instance space, each of these projections has a defined **destination** (a set name shown after the right-arrow below) for each possible value of X_4. Creating sets from all projected instances in $\mathcal{X}'$ that have the same destinations for the same values of X_4, we get

$$C_{01}(0 \rightarrow C_0, 1 \rightarrow C_1) = \{000, 001, 110, 111\}$$
$$C_{10}(0 \rightarrow C_1, 1 \rightarrow C_0) = \{010, 011, 100, 101\}$$

Note that out of four possible sets, only two were needed. We now construct the OODG recursively using the two non-empty sets C_{01} and C_{10} as our input sets. Selecting variable X_3 to delete gives us the following projection

$$\mathcal{X}'' = \{00, 01, 10, 11\}$$

Creating the appropriate sets from the projected instances in $\mathcal{X}''$ yields

$$C_{00}(0 \rightarrow C_{01}, 1 \rightarrow C_{01}) = \{00, 11\}$$
$$C_{11}(0 \rightarrow C_{10}, 1 \rightarrow C_{10}) = \{01, 10\}$$

Note that each of the two new sets implements a constant function that ignores the value of the given attribute (*i.e.*, it branches to the same node regardless of the attribute value). A level for which all implemented functions are constant implies that the variable is irrelevant. Continuing the execution yields the OODG depicted in Fig.1.

4 HOODG: A Hill Climbing Algorithm For Constructing OODGs

In this section we address the two main problems ignored in the algorithm described in the previous section:

1. Deciding where to place projected instances (Step 4 of the algorithm) for which we have more than once choice. This can happen if a projection does not have all possible augmentations because we do not have the full instance space.
2. Ordering the variable for selection in Step 2.

4.1 Placing Projected Instances

If we do not have the full instance space, there will be projections of instances for which some values of the deleted attribute will be missing (*e.g.*, we know that a projected instance must branch to some node on values 0 and 2, but do not know where it should branch on values 1 and 3). Call such projections *Incomplete Projections,* or **IPs.** A decision on where to place these instances constitutes a bias, since it determines how unseen instances will be classified.

Following Occam's razor principle, we would like to find the smallest OODG consistent with the data (since we assume no noise, we will not overfit the data). We are thus looking for a minimal set of branching nodes that "covers" all projections, *i.e.*, a minimal cover.

An IP is **consistent** with another projection, P, (at the same level of the graph) if they do not have conflicting destinations on the same value of the deleted variable. An IP **agrees** with another projection, P, if they are consistent, and all destinations defined for the IP are also defined for the projection P (note that *agrees* is an asymmetric relation).

The placement strategy used in our implementation is to start creating projection sets (branching nodes) with projections having the greatest number of known destinations (values of the deleted variable), then for projections with fewer known destinations. Each projection is placed in a projection set where it agrees with all instances whenever possible; otherwise, it is placed in a set where it is consistent with all instances, if possible; otherwise, a new projection set is created, consisting of the single projection. Ties are broken in favor of projection sets with more destinations.

4.2 Ordering the Variables

There are $n!$ possible orders in which to select the variables in Step 2 of the algorithm. Given the full instance space, it is possible to find the optimal ordering using dynamic programming by checking "only" 2^n orderings, as described in [11, 12].

In our implementation, we chose to greedily select the variable that yields the smallest width (minimal number of nodes) at the next level (equivalent to a one-ply lookahead). We consider each variable in turn, compute the width of the level that would be formed if we were to select that variable in Step 2 of the algorithm, and then select the one that minimizes the width. Tie-breaking for variables with the same resultant width favors those with the greater number of "constant" nodes, *i.e.*, nodes that ignore the variable value. This tends to favor irrelevant attributes early, increasing the ratio of the training set size to the projected instance space. Equality is possible only if each projected instance has exactly $|D|$ instances in the sets before the projection, where $|D|$ is domain size of the irrelevant variable.

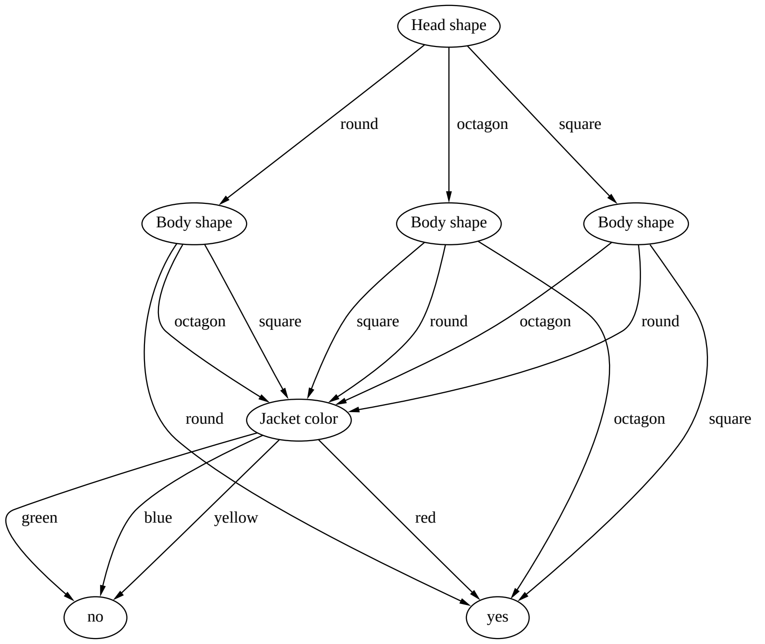
Example 2. The Monk's Problems The Monk's problems are three artificial problems that allow comparison of algorithms. In [27], 24 authors have compared 25 machine learning algorithms on these problems. In the given domain, robots have six different nominal attributes as follows:

$$\begin{aligned}
\text{Head-shape} &\in \{\text{round, square, octagon}\}. \\
\text{Body-shape} &\in \{\text{round, square, octagon}\}. \\
\text{Is-smiling} &\in \{\text{yes, no}\}. \\
\text{Holding} &\in \{\text{sword, balloon, flag}\}. \\
\text{Jacket-color} &\in \{\text{red, yellow, green, blue}\}. \\
\text{Has-tie} &\in \{\text{yes, no}\}.
\end{aligned}$$

In the first problem, Monk 1, the target concept is

$$(\text{Head-shape} = \text{Body-shape}) \text{ or } (\text{Jacket-color} = \text{red})$$

The standard training set for this problem has 124 instances out of 432 possible instances. Running our algorithm on this problem shows the significance of discovering irrelevant attributes. The three irrelevant attributes are selected first, and after constructing three levels of constant nodes, we are left with a problem of inducing an OODG for the remaining three attributes. There are 36 possible instances for the three remaining attributes, and the projections of the specific training set given in this problem yield 35 different instances (the expected number is 34.91, so this is not a fortuitous training set). The missing instance agrees with only one of the two nodes at the level at which it is projected, so we do not have a choice of where to put it. After discovering the irrelevant attributes, the problem becomes trivial. Fig.4 shows the resulting OODG, after removing constant nodes.



Target concept is: (Head-shape = Body-shape) or (Jacket-color = red).

Fig. 4. Resulting OODG for Monk's problem 1 (constant nodes removed).

In the second problem, Monk 2, the concept is

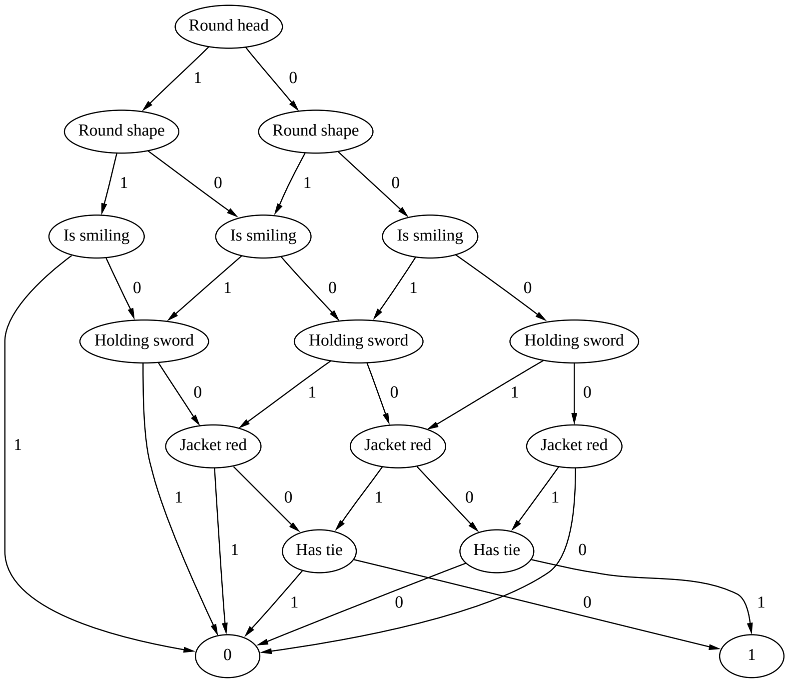
Exactly two of the attributes have their *first* value.

The standard training set for this problem has 169 instances. This problem has no irrelevant attributes which makes it very hard, given the original encoding of six nominal attributes. However, if we encode it using a *local representation,* where each attribute value is represented by one Boolean indicator variable, many attributes become irrelevant. This is the encoding scheme used by Thrun and Fahlman when neural nets were tested.

Under a local-representation encoding, there are 17 attributes, but 11 of them are irrelevant. Projecting the instances on the six relevant attributes yields 52 different instances out of 64 possible ones.

There is a subtle problem when running our hill-climbing algorithm. In the first stage, each of the 17 attributes is redundant (*i.e.*, the penultimate level has only constant nodes when each is selected), making it a candidate for being an irrelevant attribute. Intuitively, deleting any *one* such Boolean attribute still allows us to solve the problem; in the local representation, the missing attribute can be reconstructed from the other attributes, since for every variable in the original encoding, *exactly* one of the new indicator variables must be set to 1 (and the others to zero). Note of course that once one variable is deleted, other variables that were "candidates" become relevant. Our heuristic for choosing the "best" attribute is to do one level of lookahead for redundant attributes. The attribute that creates the most number of redundant attributes at the *next* level is selected.

With the lookahead procedure described above, and using local representation, HOODG correctly generalizes the training set to get 100% accuracy on the test set. Fig.5 shows the resulting OODG, which is the smallest one possible for the given target concept.



Target concept is: Exactly two of of the following attributes must have the value 1 {round-head, round-shape, is-smiling, holding-sword, jacket-red, has-tie}.

Fig. 5. Resulting OODG for Monk's problem 2

5 Experimental Results

The complexity of our algorithm as implemented is $O(ns^2 + is^2(n-1))$ per level, where i is the number of irrelevant attributes at the given level, and s is the number of projected instances at that level. Running on a a SPARCstation ELC, the execution time varies from about 3 seconds for Monk 1, 10 seconds for parity 5+5 (described below) to four and a half minutes for Monk 2 in the local representation (mostly due to the large number of irrelevant attributes).

Table 1 shows the accuracy results for some classical datasets and some artificial ones. The datasets Monk 1,2,3, and Vote, were taken from Quinlan's C4.5 data files (Monk3 has 5% noise); each has one training set and one test set. The vote database is a real-world database that includes votes for each of the U.S. House of Representatives Congressmen on the 16 key votes identified by the Congressional Quarterly Almanac Volume XL (the votes are simplified to yes, no, or unknown). The data set consists of 300 instances and the test set consists of 135 instances.

The second part of the table depicts the average of 10 runs, each with a randomly chosen training set. Each training set for Monk 1* is of size 124 (as is the original training set); each training set for Parity5, which is the XOR of 5 bits, consists of 50% of the instance space; and each training for Parity 5+5, which is the XOR of 5 bits with 5 irrelevant bits, consists of 10% of the instance space. The systems compared are ID3, C4.5, with and without grouping (-s flag), and HOODG.

Data Set	ID3	C4.5 / C4.5 (grouping)	HOODG
Monk 1	81.7%	75.7%/100.0%	100.0%
Monk 2	69.2%	65.0%/74.1%	83.1%
Monk 2 (local repr.)	86.6%	70.4%/75.9%	100.0%
Monk 3	94.4%	97.2%/100.0%	94.4%
Vote	94.1%	97.0%/93.3%	94.1%
Monk 1*	92.3% $\pm$ 4.6%	86.1% $\pm$ 3.7%/92.9% $\pm$ 7.0%	100% $\pm$ 0.0%
Parity 5 (50%)	60.6% $\pm$ 3.0%	50.0% $\pm$ 0.0%/50.0% $\pm$ 0.0%	100% $\pm$ 0.0%
Parity 5+5 (10%)	55.2% $\pm$ 4.5%	52.5% $\pm$ 4.7%/52.5% $\pm$ 4.7%	100% $\pm$ 0.0%

Table 1. Comparison of different algorithms. Results in the second part of the table are averaged over 10 runs with standard deviation after the $\pm$ sign.

It is interesting to note that while C4.5 gets 100% on the original training set for Monk 1, it has a large variance when executed on different training sets of the same size.

6 Related Work

Lee [15] introduced *binary decision programs* that are evaluated by executing a series of instructions that test a variable and make a two way branch. He showed that it is possible to represent any switching function in $O(\frac{2^n}{n})$ such instructions.

Akers [1] described binary decision diagrams, and gave a top-down procedure for building them using the Boole-Shannon expansion [3, 26]:

$$f = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0}$$

where $f|_{x_i=b}$ is the restriction, or cofactor, of the function f

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

Moret [20] gives an excellent survey of work on decision trees and diagrams, with over 100 references.

Bryant [6] introduced Ordered Binary Decision Diagrams (OBDDs), which spawned a plethora of articles and a whole subcommunity dealing with OBDDs [7, 5, 19, 13]. OBDDs are a restriction of Binary Decision Diagrams (BDDs), where a total ordering is defined over the set of variables and all paths must test variables in accordance with the given ordering. Note that OBDDs are *not* necessarily levelled. Bryant describes the advantages of OBDDs over the common representations like CNF and DNF (these advantages apply to OODGs too):

- Operations like complementation may yield exponential growth for DNF and CNF, while they do not change the size of OBDDs.
- Common operations such as reduction, $f_1 < \mathrm{op} > f_2$ (where op is any binary function), restriction, and composition, are bounded by the product of the graph sizes for the functions being operated on.
- Satisfiability testing takes constant time (check if the OBDD is the single category node **0**), while finding a satisfying assignment for n variables takes $O(n)$ Counting the number of satisfying assignments if $O(|G|)$ where $|G|$ is the size of the graph, and finding all satisfying assignments is $O(n \cdot |S_f|)$ where $|S_f|$ is the number of such satisfying assignments.

OBDDs have been used for automatically verifying finite state machines, including 64-bit ALUs, with up to 10^{120} states by representing the state space symbolically instead of explicitly [9, 8]. These applications show, at least empirically, that many functions occurring in engineering domains seem to be representable in small (polynomial) OBDD structures (and hence in OODGs).

In the computer science theory community, binary decision graphs have been called **branching programs,** and have been extensively studied in the hope of separating some complexity classes and for studying the amount of space needed to compute various functions [4]. Two important theorems tell us that an algorithm in SPACE($S(n)$) for $S(n) \geq \log n$ has a branching program complexity of at most $c^{S(n)}$ for some constant c [16], and that constant-width branching programs are very powerful, being able to accept all NC^1 languages [2].

In the machine learning community, general decision graphs were investigated by Oliver [21, 22] whose algorithm constructs the graphs top-down, by doing a hill-climbing search through the space of graphs, estimating the usefulness of each graph by Wallace's MMLP (minimum message length principle). At each stage a decision is made whether to split a leaf (and which), or whether to join to leaves. Operations that increase the message-length are never performed, hence the algorithm is guaranteed to terminate. The algorithm is (heuristically) able to overcome the replication and fragmentation problem associated with decision-trees.

Dvorak independently discovered the bottom-up technique we have used here to minimize OBDDs [10]. Although his work resembles ours, his motivation is to minimize functions with Don't Cares, while ours is to induce structures with high predictive power.

The relations between the different models, that is, OBDD, Branching Programs, and Decision Trees are summarized by Meinel in [17]. Translating Meinel results to the terms used in this paper, we get the following lemmas:

- There exists a Boolean function for which the smallest decision tree representing it has size $O(2^n)$, while there is an OODG representing it of size $O(n)$.
- There exists a Boolean function for which the smallest OODG representing it has size $O(2^{(n/\log n)})$, while there is a tree of size $O(n^2/\log n)$ representing it. (This lemma shows the different bias of the two structures.)
- There exist a Boolean function for which the smallest OODG has size size $2^{\Omega(n)}$, while an oblivious decision graph of depth linear in n that is not read-once, can represent it in $n^{O(1)}$.

An interesting point, first mentioned by Lee and Akers and studied in [19], is that a decision diagram actually represents more than one function. Entering the diagram at a different node allows *sharing* functions.

7 Future Work

Since our algorithm is essentially a hill-climbing algorithm, it may not find a global minimum. Researchers working on OBDDs (cf. [18, 14]) have experimented with exchanging variables after building the graph, and have achieved good results.

Deeper lookahead for variable selection is an obvious possible extension, especially since one motivation for growing the graph from the bottom is the asymmetric shape (the kite shape depicted in Fig.2) bounding the OODG.

We have shown promising preliminary results of using this structure with a simple hill-climbing heuristic. Looking further down the road, we need a method of dealing with noise, possibly by pruning the OODG. We need to extend the variable domains from discrete to real-valued, and to conduct more experiments on the generalization power.

Acknowledgements We would like to thank Nils Nilsson and Yoav Shoham for their continued support for this idea, and for supporting the MLC++ project at Stanford. Thanks to everyone who contributed to MLC++, especially George John, Richard Long, David Manley, Ofer Matan, and Karl Pfleger. Thanks to Ronen Brafman, Pat Langley, John Oliver, Ron Rymon, and Tomas Uribe for their comments on the first draft of this paper, and to James Kittock for carefully

reading drafts of this paper. Our implementation was written using the MLC++ library, and the OODGs depicted in this paper are actual outputs generated by library routines that interface with AT&T's **dot** program written by Koutsofios and North.

References

1. Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.
2. David A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.
3. George Boole. *An investigation of the laws of thought, on which are founded the theories of logic and probabilities.* London, Walton and Maberly; Macmillan and Co., 1854. Reprinted by Dover Books, New York, 1954.
4. Ravi B. Boppana and Michael Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier, 1990.
5. Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference. Proceedings*, pages 40–45, 1990.
6. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on computers*, C-35(8):677–691, 1986.
7. Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
8. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference. Proceedings*, pages 403–407, 1991.
9. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science.*, pages 428–439. IEEE Comput. Soc. Press, 1990.
10. Vaclav Dvorak. An optimization technique for ordered (binary) decision diagrams. In P. Dewilde and J. Vandewalle, editors, *Compeuro Proceedings. Computer Systems and Software Engineering*, pages 1–4. IEEE Comput. Soc. Press, 1992.
11. Steven J. Friedman and Kenneth J. Suppowit. Finding the optimal variable ordering for binary decision diagrams. In *24th ACM/IEEE Design Automation Conference*, pages 348–355, 1987.
12. Steven J. Friedman and Kenneth J. Suppowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions On Computers*, 39(5):710–713, 1990.
13. Masahiro Fujita, Hisanori Fujisawa, and Jusuke Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, 1993.
14. Nagisa Ishiura, Hiroshi Sawada, and Shuzo Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *IEEE International Conference On Computer-Aided Design. Digest of Technical Papers*, pages 472–475. IEEE Comput. Soc. Press, 1991.
15. C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

16. William J. Masek. A fast algorithm for the string editing problem and decision graph complexity. Master's thesis, Massachusetts Institute of Technology, 1976.
17. Christoph Meinel. Branching programs — an efficient data structure for computer-aided circuit design. *Bulletin of the European Association For Theoretical Computer Science*, 46:149–170, 1992.
18. Shin-ichi Minato. Minimum-width method of variable ordering for binary decision diagrams. *IEICE Transactions On Fundamentals of Electronics, Communications and Computer Sciences*, E75-A(3):392–399, 1992.
19. Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *27th ACM/IEEE Design Automation Conference. Proceedings*, pages 24–28, 1990.
20. Bernard M. E. Moret. Decision trees and diagrams. *ACM Computing Surveys*, 14(4):593–623, 1982.
21. J.J. Oliver, D.L. Dowe, and C.S. Wallace. Inferring decision graphs using the minimum message length principle. In A. Adams and L. Sterling, editors, *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pages 361–367. World Scientific, Singapore, 1992.
22. Jonathan J. Oliver. Decision graphs — an extension of decision trees. In *Proceedings of the fourth International workshop on Artificial Intelligence and Statistics*, pages 343–350, 1993.
23. Giulia Pagallo and David Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5:71–99, 1990.
24. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. Reprinted in Shavlik and Dietterich (eds.) Readings in Machine Learning.
25. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, Los Altos, California, 1992.
26. C. E. Shannon. The synthesis of two-terminal switching circuits. *The Bell System Technical Journal*, 28(1):59–98, 1949.
27. S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski andS.E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Weldel, W. Wenzel, J. Wnek, and J. Zhang. The monk's problems: A performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon University, 1991.
28. Ingo Wegener. *The Complexity of Boolean Functions*. B. G. Tuebner, 1987.