Decision Trees and Diagrams

Article *in* ACM Computing Surveys · December 1982

DOI: 10.1145/356893.356898 · Source: DBLP

CITATIONS
331

READS
1,817

1 author:

Bernard M. E. Moret
École Polytechnique Fédérale de Lausanne
224 PUBLICATIONS **7,263** CITATIONS

SEE PROFILE

DECISION TREES AND DIAGRAMS Decision trees and diagrams (also known as sequential evaluation procedures) have widespread applications in databases, decision table programming, concrete complexity theory, switching theory, pattern recognition, and taxonomy; in short, wherever discrete functions must be evaluated sequentially. This tutorial survey establishes a common framework of definitions and notation, reviews the contributions from the main fields of application, presents recent results and extensions, and discusses areas of ongoing and future research.

The research leading to this work was supported by the Office of Naval Research, Arlington, Virginia, under contract No. 0014-78-C-0311.

CR Categories and Subject Descriptors: B.6.1 [**Logic Design**] Design Styles, B.6.3 [**Logic Design**] Design Aids switching theory, D.1.m [**Programming Techniques**] Miscellaneous, F.2.2 [**Analysis of Algorithms and Problem Complexity**] Nonnumerical algorithms and problems computations on discrete structures, G.2.2 [**Discrete Mathematics**] Graph Theory trees, H.2.4 [**Database Management**] Systems query processing, I.2.8 [**Artificial Intelligence**] Control Methods and Search, I.5.1 [**Pattern Recognition**] Models, I.5.2 [**Pattern Recognition**] Design Methodology, J3 [**Life and Medical Sciences**] biology, health

General Terms: algorithms, design, theory

Additional Key Words and Phrases: atomic digraph, binary identification, Boolean graph, decision program, decision table, diagnostic key, diagnostic table, evaluation, exhaustive function, feature selection, heuristics, hierarchical classifier, multiplexer network, multistage testing, NP-complete problem, sequential evaluation procedure, table splitting, taxonomy, test selection

Author's address: Department of Computer Science, The University of New Mexico, Albuquerque, N.M. 87131.

DECISION TREES AND DIAGRAMS

Bernard M. E. Moret

The University of New Mexico

A decision tree or diagram is a model of the evaluation of a discrete function, wherein the value of a variable is determined and the next action (to choose another variable to evaluate or to output the value of the function) is chosen accordingly. Decision trees find many applications in decision table programming [Shaw71, Pooc74, Metz77], data bases [Wong76, Hana77], pattern recognition [Haus75, Bell78], taxonomy and identification [Jard71, Mors71, Gare72a, PayR80, Will80], machine diagnosis [Klet60, Chan70], switching theory [Lee59, Thay81a], and analysis of algorithms [Weid77]. More recently, they have been proposed as implementation-independent models of discrete functions with a view to the development of new testing methods [Aker79, More81a] and complexity measures [More80a]. Due to this broad applicability, results about decision trees are dispersed throughout the literature in fields such as biology, computer science, information theory, and switching theory; moreover, there is no common notation or set of definitions. Therefore, this article begins by establishing a framework of notation and definitions which introduce decision trees and diagrams and the various measures associated with them. Particular attention is paid to the problem of constructing decision trees and diagrams from function descriptions and evaluating their efficiency. The complexity of such constructions is detailed and repercussions on circuit or program design analyzed. A survey of the main fields of application and related results follows. Recently proposed extensions (to include diagram composition and recursion) and applications (e.g., to system testing) are then discussed. The article concludes with an assessment of known results and suggestions for future research. The emphasis throughout this exposition is on Boolean functions, since they find many more applications and are more readily understood than general discrete functions. The presentation alternates formal exposition, examples, and discussion; complex proofs are avoided (the reader will find them in the references) and the mathematical content is kept to the minimum necessary for clarity and conciseness. In particular, the first section introduces all necessary mathematical and other background, so that the paper should be accessible to any reader with a mathematical or algorithmic bent. The intent is to cover the breadth of the field, unify terminology, convey the import of the main results, and act as a guide to the literature, for which last purpose a representative, rather than exhaustive, bibliography is provided. As such, this survey should be of interest to both practitioners and researchers in the areas mentioned above. Since the evaluation of Boolean functions, the programming of decision tables, and the identification of unknown objects (biological specimens, system faults, etc.) are among the most important applications of decision trees and diagrams, we provide a succinct review of the terminology and basic concepts of Boolean functions, decision tables, and identification problems. Readers who feel comfortable with these topics may wish to skip to Section 3. Only a very brief review is provided; for more details, the reader is referred to Davi80 on discrete functions and to Harr65 on Boolean functions. By discrete

function, we mean a (partial) function of discrete variables, $f(x_1,\ldots,x_n)$, where each variable, x_i, takes exactly m_i values, which we choose to denote $0,\ldots,m_i-1$. A discrete function is constant if and only if (iff) it assumes the same value wherever it is defined; it is null if it is not defined in any point of its domain, completely specified if it is defined everywhere. When a variable is evaluated, say $x_i = k$, we are left with the restriction, $f(x_1,\ldots,x_{i-1},k,x_{i+1},\ldots,x_n)$, which we denote $f_{vbar\ x_i=k}$. A variable, x_i, is redundant iff $f_{vbar\ x_i=0} = \cdots = f_{vbar\ x_i=m_i-1}$, where two functions are equal if they have the same domain and codomain and assume the same value wherever they are both defined; a function without redundant variables is called intrinsic. Finally, a variable, x_i, is termed indispensable iff it is not redundant in any restriction resulting from the evaluation of any subset of the variables $\{x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n\}$. (This implies that a function can never be evaluated at any point without knowledge of the values of all indispensable variables.) A Boolean function of n variables is a discrete function, $f\colon \{0,1\}^n\ rarrow\ \{0,1\}$, where $\{0,1\}^n$ denotes the n-fold cartesian product of $\{0,1\}$, that is, the set of all binary n-tuples. Each n-tuple, $(x_1,\ldots,x_n)$, mapped to 1 by the function is a minterm of the function. A Boolean function can be specified by describing the mapping (giving its "truth table") or by listing its minterms and those points at which it is not defined (so-called "don't care" conditions); it can also be represented by a Boolean formula, usually in terms of the three operations of disjunction (+), conjunction (·), and complementation (-). A Boolean function of n variables can be expressed in terms of two functions of $n-1$ variables by means of Shannon's expansion theorem: $f(x_1,\ldots,x_n) = \overline{x_i}\cdot f_{vbar\ x_i=0} + x_i \cdot f_{vbar\ x_i=1}$, for each choice of x_i.

Example 1: Consider the Boolean function of three variables, $f(x_1, x_2, x_3)$, given by the mapping:

f:	(0,0,0)	$rarrow$	0	(1,0,0)	$rarrow$	0
	(0,0,1)	$rarrow$	0	(1,0,1)	$rarrow$	0
	(0,1,0)	$rarrow$	1	(1,1,0)	$rarrow$	0
	(0,1,1)	$rarrow$	1	(1,1,1)	$rarrow$	1

Since every point in the domain is assigned a value, the function is completely specified. Other representations for f are the list of its minterms: {(0,1,0),(0,1,1),(1,1,1)}, or a Boolean formula: $\overline{x_1}x_2\overline{x_3} + \overline{x_1}x_2x_3 + x_1x_2x_3$. The latter formula is equivalent to the list of minterms; it can be simplified to yield the minimum expression: $\overline{x_1}x_2 + x_2x_3$. It is easily verified that the function is intrinsic; expanding it around x_2 yields $f = \overline{x_2}\cdot 0 + x_2\cdot(\overline{x_1} + x_3)$. *box*

The terminology used in the following is that of Metz77; other general references are Shaw71 and Pooc74. A decision table is an organizational or programming tool for the representation of discrete functions. It can be viewed as a matrix where the upper rows specify sets of conditions and the lower ones sets of actions to be taken when the corresponding conditions are satisfied; thus each column, called a rule, describes a procedure of the type "if conditions, then actions."

Example 2: The following decision table describes how to spend a Saturday afternoon in spring. It has two condition

rows, three action rows, and four rules; the first condition is a binary variable (taking values "yes" or "no") while the second is a ternary variable (taking values "calm", "breezy", or "windy"). According to normal practice [Metz77], condition and action names are used as labels on appropriate rows and a rule is specified by entering values in the condition rows (or blanks, for don't care conditions) and X's (meaning "execute") in the action rows.

Raining?	yes	no	no	
Wind condition		breezy	calm	windy
Clean basement	X			X
Spade garden			X	
Fly kite with children		X		

The four rules can be read as: "if it is raining, then clean the basement"; "if it is breezy and not raining, then fly kite with children"; "if it is calm and not raining, then spade the garden"; "if it is windy, then clean the basement."*box* A pair of rules overlaps if a combination of condition values can be found that satisfies the condition sets of both rules. If two overlapping rules specify different actions, they are called inconsistent and the table is said to be ambiguous; if they specify identical actions, they are termed redundant. Decision tables described so far are in so-called extended-entry form. Often, however, it is required that all conditions be Boolean variables; this gives rise to limited-entry decision tables. Although most such tables are set up in limited format from their conception, it may be necessary to convert extended-entry tables to limited-entry format; this is done by using one Boolean variable for each value (but one) of the multivalued variable to be replaced [Pres65]. This process results in tables where entries in one condition row often imply (absent) entries in others; such implied entries can also be present in any decision table and give rise to apparent (but non-existent) ambiguity. Since the implications result from purely semantic considerations, they cannot be detected by an automatic processor, so that they must be explicitly specified. The impossible combinations of conditions will then be treated as inputs with unspecified mapping.

Example 3: In the decision table of Example 2, rules 1 and 4 overlap because they are both applicable when it is raining and windy. Since they specify the same action set, they are redundant, and since no other rules overlap, the table is unambiguous. The same table, converted to limited-entry format, is shown below; it still has three action rows and four rules, but now has three condition rows. Implied entries are shown in parentheses; their absence, while

not confusing to a human, would induce an automatic processor to decide that the second and third rules are inconsistent, since both could apparently apply when it is not raining and it is calm and breezy.

Raining?	yes	no	no	
Calm?		(no)	yes	(no)
Breezy?		yes	(no)	(no)
Clean basement	X			X
Spade garden			X	
Fly kite with children		X		

It is noted that the specification of implied entries in the table is insufficient: while it identifies the impossible condition set (no,yes,yes), it fails to identify the equally impossible set (yes,yes,yes), which will be erroneously included in the first rule. This suggests that logical inconsistencies be separately listed [King73]; for instance, the above table would be supplemented by the logical expression NOT (breezy AND calm).*box*

It should now be clear that an unambiguous extended-entry decision table is a special case of a partial function of multivalued variables, where the conditions correspond to the variables and the action sets to the function values. In particular, a complete decision table (one which has an applicable rule for every combination of conditions), corresponds to a completely specified function and a limited-entry decision table corresponds to a function of binary variables. An ambiguous decision table can be modelled by a relation, as discussed later. A sequential evaluation procedure for a decision table is then of particular importance, since it corresponds to an implementation, usually in software, of the decision table; indeed, the importance of the limited-entry format is in good part due to the ease of programming binary decisions (by if-then-else constructs) [Metz77]. Consider the situation where an unknown event or specimen is to be classified into one of a finite number of known categories, based upon the outcome of a number of tests. (This is a special case of the concept of questionnaires [Pica72].) Such identification problems arise in biology, medical diagnosis, machine trouble-shooting, and numerous pattern recognition applications. A binary identification problem includes only binary tests. As defined in Gare72a, a binary identification problem consists of a finite set of **objects**, $(O_1, \ldots, O_n)$, which represents the universe of possible identifications, and a finite set of **tests**, $(T_1, \ldots, T_m)$,

each of which is a function from the set of objects to the set {*yes,no*} (that is, each test applied to a specific object gives a specific *yes/no* answer). In a simple binary identification problem, all tests are of the form "is the unknown object of type i ?", that is, their outcome is *no* for all but one object [Gare72b]. An optional probability distribution can be specified on the set of objects, giving the a priori probability that an unknown object will be identified as each object in the set. In practical situations, the number of tests and the number of objects are often in the same range (even though, in theory, the number of tests could be arbitrarily larger than that of objects and the number of objects could grow as an exponential function of the number of tests). In our terminology, the tests are binary variables and the objects are values of a partial bijective function from the variables' space to the set of objects. In biology, tests are also known as characters or conditions and objects as taxa or formae, while the specification of an identification problem is known as a diagnostic table; in pattern recognition, tests would be known as features and objects as classes; in questionnaire theory, the tests are questions and their outcomes are responses.

Example 4: Consider the identification problem given by a set of five objects, {a,b,c,d,e}, a set of four tests, {T_1={a}, T_2={a,b}, T_3={a,b,c}, T_4={a,b,c,d}}. The same problem can be represented as a diagnostic table as below, where the rows correspond to objects and the columns to tests.

	T_1	T_2	T_3	T_4
a	yes	yes	yes	yes
b	no	yes	yes	yes
c	no	no	yes	yes
d	no	no	no	yes
e	no	no	no	no

In our terminology, we have a partial bijective function of four binary variables, defined in five points and given by the following mapping:

(yes,yes,yes,yes) *rarrow* a

(no,yes,yes,yes) *rarrow* b

(no,no,yes,yes) *rarrow* c

(no,no,no,yes) *rarrow* d

(no,no,no,no) *rarrow* e

box

This formulation provides a clean, but very much simplified model of the general identification problem. In pattern recognition applications, the test results are generally not as clearly defined; instead, each test may take any of the possible values, as specified by a discrete probability function. The same problem arises in biology (where it is

known as probabilistic identification); however, since the probability distribution is often unknown, a confidence threshold is normally used, which dichotomizes test outcomes as "known" (taking a specific value) or "variable" (susceptible of taking any value). In the following, we shall first look at Garey's model, then generalize results to include the probabilistic identification model. A decision tree (or diagnostic key, as known in many identification applications) can be regarded as a deterministic algorithm for deciding which variable to test next, based on the previously tested variables and the results of their evaluation, until the function's value can be determined. Several constraints are placed upon such an algorithm, all designed to prevent clearly redundant testing. The following formal definition of a decision tree is taken from More80b.

Definition 1: Let $f(x_1, \ldots, x_n)$ be a (partial) function of discrete variables. If f is constant or null, then the decision tree for f is composed of a single leaf labelled by the constant value or by the null symbol. Otherwise, for each x_i, $1 \leq i \leq n$, such that at least two restrictions, say $f_{vbar\ x_i=k_1}$ and $f_{vbar\ x_i=k_2}$, are not null, f has one or more decision trees composed of a root labelled x_i, and m_i subtrees which are decision trees corresponding to the restrictions $f_{vbar\ x_i=0}, \ldots, f_{vbar\ x_i=m_i-1}$, in that order. *box*

This recursive definition closely parallels the conventional definition of ordered trees, such as binary trees [Knut73]; it defines decision trees as rooted, ordered, vertex-labelled trees where each node has either m_i children for some i, $1 \leq i \leq n$, or none (and is a leaf). To an extent, this definition prevents redundant testing: a variable is tested only once on any path; no variable can be tested which would result in all restrictions but one being null; and no more testing can take place as soon as the function has been reduced to a constant. The evaluation of a discrete function represented as a decision tree starts by ascertaining the value of the variable associated with the root of the tree. It then proceeds by repeating the process on the k-th subtree, where k is the value assumed by the root variable, until a leaf is reached; the label of the leaf gives the value of the function.

Example 5: The Boolean function of Example 1 was given by the formula: $f(x_1, x_2, x_3) = \overline{x_1}x_2 + x_2x_3$. A possible decision tree for that function is shown in Figure 1. Since decision trees are ordered, the left subtree of a node corresponds to the node's variable evaluating at 0, the right subtree to the variable evaluating at 1. Thus the left subtree of the root corresponds to the restriction $f_{vbar\ x_1=0} = x_2$, and the right subtree corresponds to the restriction $f_{vbar\ x_1=1} = x_2x_3$. Evaluation on the tree for the triple of values (1,0,0) starts by examining x_1; on finding it to be 1, it proceeds to the right subtree, there to evaluate x_3. Since x_3 is found to be 0, the left subtree is next used, thereby encountering a leaf and terminating the evaluation, having used only two of the three variables; the label of the leaf is the value of the function, i.e., f(1,0,0) = 0. It is noted that a decision tree for a Boolean function is an explicit illustration of Shannon's expansion; the tree of Figure 1 represents the expansion: $f(x_1, x_2, x_3) = \overline{x_1} \cdot [\overline{x_2} \cdot 0 + x_2 \cdot 1] + x_1 \cdot [\overline{x_3} \cdot 0 + x_3 \cdot (\overline{x_2} \cdot 0 + x_2 \cdot 1)]$. *box*

We note that the same subtree may occur on several branches of the tree, in which case it may be desirable to use only one copy of that subtree by transforming the decision tree (through a process known as reticulation [PayR77]) into a simple decision diagram, which has the structure of a rooted, directed, acyclic (hyper)graph. In the case of Boolean functions, further requiring that there be only one leaf labelled 1 (the "finish" node) yields a free Boolean graph. Of course, we can choose which identical subtrees to merge, if any; in particular, every decision tree is a (simple) decision diagram. To every simple decision diagram there corresponds a unique decision tree; moreover, the paths in the diagram are in one-to-one correspondence with those in the tree. Conversely, to every decision tree for a completely specified function there corresponds a unique "minimal" diagram, that is, one in which every possible merge has been accomplished. Reticulation sometimes also merges non-identical subtrees while preserving the identity of the function; in such cases, it may happen that a variable occurs more than once along a path from the root to a leaf. Such non-simple decision diagrams may further decrease the total number of nodes required; however, the second test of a variable is redundant and thus detracts from the diagram's "efficiency". Decision diagrams (and their corresponding trees) can easily be programmed. Lee59 calls the result (simple) decision programs and has suggested a universal instruction type which implements the evaluation process taking place at an internal node: L: $i, g_0, \ldots, g_{m_i-1}$, where L is a label, i identifies variable x_i, and g_k, used only when $x_i = k$, is either a value (if the restriction for $x_i = k$ is a constant) or a label. Such an instruction is executed by testing variable x_i and upon finding its value, say $x_i = k$, taking the corresponding action, g_k, that is, either transferring control to the instruction labelled g_k or assigning to the function the value g_k. Thus to each node of the diagram there corresponds one instruction in the program. Cern79b has investigated a special-purpose architecture for the execution of such programs. Decision trees and diagrams for Boolean functions find yet another implementation, this time in hardware as multiplexer trees and networks [Cern79a, Thay81a]. In a multiplexer tree (network), each internal tree (diagram) node is represented as a 2-1 multiplexer controlled by the node variable and each leaf is implemented as a constant logical value (wired-at-0 or wired-at-1); the interconnection scheme is that of the decision tree (diagram). The evaluation of a function then proceeds from the "leaves" (the constant values) to the "root" multiplexer; the function variables, used as control variables, select a unique path from the root to one leaf and the value assigned to that leaf propagates along the path to the output of the "root" multiplexer.

Example 6: Consider the tree of Example 5. It is itself a diagram; merging the identical subtrees rooted in nodes labelled x_2 and the identical leaves results in the minimal free Boolean graph pictured in Figure 2. This diagram can be implemented by the decision program given below (where letters are used for labels to distinguish them from values).

```
Start:   1, A, B
   A:    2, 0, 1
```

B: 3, 0, A

Figure 3 depicts an equivalent multiplexer network. We note that, as a consequence of our definitions, the number of multiplexers used in a network is precisely the number of instructions of an equivalent decision program; similarly, the maximum delay through a network is proportional to the maximum execution time of an equivalent program, both being dependent upon the length of the longest path through the diagram.*box* Since the root of each subtree can be labelled with any (up to the restrictions of Definition 1) of the untested variables, the number of possible decision trees for a given function is in general very large (and that of possible decision diagrams even larger). For instance, the function of Example 5 has ten distinct decision trees, as shown in Figure 4. In fact, a completely specified Boolean function of n variables can have up to $N_T(n) = \prod_{i=0}^{n-1}(n-i)^{2^i}$ distinct decision trees. Indeed, n choices are possible for the root, followed by $n-1$ choices on each of the two subtrees, or $(n-1)^2$ choices; in general, up to $(n-k)^{2^k}$ choices are possible at depth k. This corresponds to the recurrence relation: $N_T(n) = n \cdot (N_T(n-1))^{sup2}$, which shows that $N_T(n)$ grows faster than 2^{2^n}. The first few values of $N_T(n)$ are listed in Table 1.

n	$N_T(n)$	n	$N_T(n)$
1	1	6	$1.65 \cdot 10^{sup1sup3}$
2	2	7	$1.91 \cdot 10^{sup2sup7}$
3	12	8	$2.91 \cdot 10^{sup5sup5}$
4	576	9	$7.64 \cdot 10^{sup1sup1sup1}$
5	1658880	10	$5.84 \cdot 10^{sup2sup2sup4}$

Not all tree or diagram representations of a function are equally desirable. Thus, several criteria have been developed in order to select an appropriate representation; such criteria attempt to measure important properties of decision trees and diagrams such as their implementation and usage costs. In the most general case, each variable has an associated testing cost, which measures the expense (e.g., in time) incurred each time that variable is evaluated, and a storage cost, which measures the expense (e.g., in memory) due to the presence of each test node labelled by that variable. In addition, a probability distribution is often specified on the variables' space, which can be assumed uniform if not otherwise known. These data allow the computation of the following measures. (These and other criteria are discussed in depth in More81b.)

Definition 2: The worst-case testing cost, h, is the maximum path testing cost. When all testing costs are unity, h reduces to the worst-case number of tests, that is, the height of the tree or diagram. The expected testing cost, E, is the expected value of the path testing cost, where the probability of a path is the sum of the probabilities of all the combinations of variables' values that select that path. When all testing costs are unity, E reduces to the expected number of tests. The tree storage cost, α, is the sum of the storage costs of the internal nodes of the tree. When all costs are unity, α reduces to the number of internal nodes of the tree. The diagram storage cost, β, is the sum of the storage costs of the internal nodes of the minimal diagram. *box* In the case of unity costs and uniform probability distribution, the only datum needed to compute the first three measures (those applicable to trees) is the number of leaves at each level of the tree. Thus a decision tree for a function of n variables can be characterized by an $(n+1)$-tuple, the leaf profile [More80a], $(\lambda_0, \ldots, \lambda_n)$, where λ_i is the number of leaves at depth i. The leaf profile induces a lexicographic ordering on decision trees, thereby giving rise to two additional measures: the maximum profile, which ranks as "best" that tree which is largest in lexicographic order (on the grounds that leaves should be encountered as soon as possible); and the minimum reverse profile, which ranks as "best" that tree which is smallest in reverse lexicographic order (on the grounds that the number of long paths should be minimized).

Example 7: Assume the following storage costs, s, testing costs, t, and probability distribution, p, for the function of Example 5:

s: x_1 *rarrow* 1 x_2 *rarrow* 2 x_3 *rarrow* 3

t: x_1 *rarrow* 1 x_2 *rarrow* 2 x_3 *rarrow* 6

p: $(0,0,0)$ *rarrow* 0.10 $(1,0,0)$ *rarrow* 0.05

$(0,0,1)$ *rarrow* 0.15 $(1,0,1)$ *rarrow* 0.05

$(0,1,0)$ *rarrow* 0.05 $(1,1,0)$ *rarrow* 0.25

$(0,1,1)$ *rarrow* 0.20 $(1,1,1)$ *rarrow* 0.15

Figure 5 shows the tree of Figure 1 with its node probabilities. The expected testing cost, E, of the tree is $E = (0.25 + 0.25) \cdot (1 + 2) + 0.3 \cdot (1 + 6) + (0.05 + 0.15) \cdot (1 + 6 + 2) = 5.4$. The tree's profile is (0,0,3,2) and its various measures are listed below.

measure	value
h (worst-case testing cost)	9
height (worst-case number of tests)	3
E (expected testing cost)	5.4

expected number of tests	2.2
α (storage cost)	8
node count	4

box Finally, when ascertaining the value of a variable requires a costly apparatus (or subroutine), it may be desirable to minimize the total cost incurred through the acquisition of such apparatus for each variable used in the tree; we shall call this criterion the total acquisition cost. Minimizing this cost is a common problem in biological identification [PayR80, Will80], where the number of tests often exceeds the number of taxa; the objective is to find the smallest subset of tests that still separates all taxa, which corresponds to minimizing the total acquisition cost when all tests have unity acquisition costs. Clearly, such a cost is fixed (and maximal) for intrinsic functions, since all variables must be tested, and thus evaluated at some point or other in the tree. In fact, this cost is better associated with the functions rather than the trees. In the simplified model of binary identification expounded by Gare72a, there is exactly one combination of test values associated with each object. Therefore decision trees for such problems have a fixed number of leaves (one per object) and thus of internal nodes (since the number of internal nodes of a binary tree is one less than the number of its leaves), so that their storage cost is simply equal to one less than the number of objects when all costs are unity. Moreover, since no two leaves are identical, there can be no common subtrees, so that decision diagrams for identification problems are decision trees. In the even simpler case of simple binary identification, a *yes* answer immediately identifies the object and so terminates the evaluation, so that at most one path (that corresponding to the "no" answer) leads from a test to another; this corresponds to a degenerate tree with a number of internal nodes equal to its height. Only one optimization criterion, the expected testing cost, is applicable, and the optimization can be done step-by-step using a simple ordering of tests in terms of their cost to probability ratio [John56, Ries63, Slag64, Gare72b]. Similar conditions arise when a Boolean function is evaluated in a linear (as opposed to tree-structured) sequence [Hana77].

Example 8: Two possible decision trees for the binary identification problem of Example 4 are illustrated in Figure 6. Both trees have a storage cost of 4; their other measures are listed below.

tree	leaf profile	height	expected number of tests
left	(0,0,3,2,0)	3	2.4
right	(0,1,1,1,2)	4	2.8

box In most applications, decision trees and diagrams must be constructed from function descriptions. Since, as previously observed, numerous tree representations can be built, with varying usage costs, we naturally strive to construct a decision tree which optimizes a suitable measure. This endeavor raises several questions. Since the choice of an optimization criterion can be difficult, can a tree be constructed

which simultaneously optimizes several measures? How difficult is the optimization task for each criterion? If constructing an optimal tree is too time-consuming, are there fast heuristic methods which build acceptable suboptimal trees; if so, how good are those methods? This section answers each question in turn and provides a survey of optimization methods. In order to answer the first question, we examine the relationships between measures. We shall say that two optimization criteria are compatible if, for every function in a given family, at least one tree can be constructed which satisfies both criteria. More81b has shown that, even if we restrict our attention to the family of Boolean functions with uniform costs and probabilities (the case that is least conducive to incompatibilities), all criteria are pairwise incompatible, with two exceptions: the minimum height is a special case of the minimum reverse profile; and the exact relationship of the number of diagram nodes with the number of tree nodes is as yet unknown. (However, it is easily shown that the minimization of one does not necessarily result in the minimization of the other.) In the more general case of discrete functions with non-uniform costs and probabilities, all criteria are pairwise incompatible [More81b]. Thus it appears that the six measures defined on decision trees and diagrams are essentially independent, so that we are indeed faced with a problem of choice. In order to gather more information about the possible choices, we now address the second question. The problem of constructing optimal decision trees and diagrams has been addressed by many researchers using branch-and-bound techniques [ReiL66, ReiL67, Brei75b] and dynamic programming [Gare72a, Misr72, Meis73, Baye73, Schu76, PayH77, Mart78]. In the following, we review those and more specialized techniques. Each method is first introduced and its salient characteristics mentioned; a more detailed explanation follows, which the less mathematically inclined reader may wish to skip. Dynamic programming is of particular interest as all of our tree measures obey the "principle of optimality", that is, are such that an optimal solution can be built from optimal subsolutions. This is the case because the building of a decision subtree for each restriction is a separate problem which can be optimally solved independently of the others. This also tells us that our sixth measure, the diagram cost, does not obey the principle of optimality, since subdiagrams often overlap: the resulting interaction destroys the independence of the subproblems. The general algorithm builds the optimal tree from the leaves up by identifying successively larger optimal subtrees (one for each combination of tested and untested variables). This approach is embodied in an algorithm designed to convert limited-entry decision tables to decision trees with minimal expected testing cost [Baye73]. This solution was independently discovered by Schu76, who generalized it to extended-entry decision tables; a refined version, using some game tree heuristics, was recently published [Mart78]. A closely related procedure was developed for use in pattern recognition to minimize the worst-case or the expected testing cost of binary decision trees [Meis73, PayH77]. The earliest version of the algorithm appears due to Gare70 (see also Gare72a, Misr72) in the context of binary identification problems. For a function of n k-ary variables, the algorithm requires a number of operations proportional to $n \cdot k \cdot (k+1)^{n-1}$. Since a complete specification of the function requires $S = k^n$ items of information, the algorithm takes $O(S^{\log_k(k+1)} \cdot \log S)$ time for completely specified k-ary functions and is thus fairly efficient. For partial functions,

in particular identification problems, however, the input size may be much smaller: a binary identification problem with m tests and n objects requires the specification of m items each of size n (the answer to each test for each object) for an input size of $S = m \cdot n$. In this case, the algorithm may require time exponential in the input size, thereby being very inefficient. In fact, binary identification problems appear to be intrinsically "hard", that is, there is considerable evidence that no algorithm can be developed for them which would not require exponential time*. Technically, they are NP-hard [Gare79] problems, as proved in Hyaf76, Love79; for a detailed discussion of the exact complexity, the reader is referred to More81b. We now examine in more detail how the dynamic programming method is applied to the optimization of decision trees and provide an example; the less mathematically inclined reader may wish to skip to the beginning of the next section. If variable x_i, with testing cost t_i and storage cost s_i, is tested at the root of a decision tree for the function $f(x_1, \ldots, x_n)$, then the optimal values for the first three measures of Definition 2 are:

$$h_{\min}(f) = t_i + \max\{h_{\min}(f_{vbar\ x_i=0}), \ldots, h_{\min}(f_{vbar\ x_i=m_i-1})\},$$

$$E_{\min}(f) = t_i + \sum_{j=0}^{m_i-1} p(x_i = j) \cdot E_{\min}(f_{vbar\ x_i=j}), \quad \alpha_{\min}(f) = s_i + \sum_{j=0}^{m_i-1} \alpha_{\min}(f_{vbar\ x_i=j}), \tag{4}$$

where $p(x_i = j)$ denotes the probability that x_i takes on the value j. Similarly, the leaf profile of this tree is obtained by summing, component by component, the leaf profiles of its subtrees, then introducing a additional first component, set to 0. In a decision diagram, however, the subdiagrams representing the various restrictions usually overlap, so that the storage cost of a function is not directly related to the storage costs of its restrictions. This shows that five of our six measures indeed obey the principle of optimality. The algorithm will generate all possible restrictions of the function. For a function of k-ary variables, each variable can be in any of $(k+1)$ conditions (k values and the untested state) so that there are $(k+1)^n$ distinct combinations; generating them from the bottom k^n leaves (all variables tested) to the unique top node (all variables untested) requires a number of steps equal to: $\sum_{i=0}^{n} (n-i) \cdot \binom{n}{i} \cdot k^{n-i} = n \cdot k \cdot (k+1)^{n-1}$, since a node with i untested variables has $n-i$ possible parents (each with one more untested variable) and can be chosen in $\binom{n}{i} \cdot k^{n-i}$ ways. Therefore, using the "big Oh" notation of algorithm analysis [Weid77], the algorithm takes $O(n \cdot k \cdot (k+1)^{n-1})$ time. A restriction with $n-i$ untested variables determines a subspace of k^i points, which we shall call an i-**subcube**. The algorithm starts by considering all 0-subcubes (that is, all points in the variables' space), then forms all possible 1-subcubes by merging k 0-subcubes, in effect letting one variable be undetermined (so that a unique variable is associated with each merging). This process continues, forming all i-subcubes by merging $(i-1)$-subcubes, until the final n-subcube (the complete space) is formed. Each subcube is identified by an n-tuple of values, $(i_1, \ldots, i_n)$, where i_j is X if the j-th variable is untested at that node, and is the variable's value otherwise. For instance, the 0-subcubes identified by (0,1,…,1), (1,1,…,1),…,$(k-1,1,\ldots,1)$ can be merged into the 1-subcube given by (X,1,…,1) by letting variable x_1 be untested. Thus the process builds a lattice of $(k+1)^n$ nodes with $n \cdot k \cdot (k+1)^{n-1}$ edges. It is noted that the same i-sub-

cube can be formed by merging one of i distinct k-tuples of $(i-1)$-subcubes. As the lattice is built, each node (i.e., each subcube) is assigned a cost and a value; if we are interested in the expected testing cost, the probability of each node is also computed. The probability of an i-subcube is just the sum of the probabilities of the k merged $(i-1)$-subcubes; the value of the function for an i-subcube is that of the k merged $(i-1)$-subcubes, if their function values were identical, and is "?" (a special symbol indicating that the value is a non-constant function) otherwise. One easily verifies that these quantities are well defined, that is, are independent of the choice of the merged subcubes. Finally, the cost assigned to an i-subcube is the minimum cost of merging, where the merging of k $(i-1)$-subcubes has cost 0 if all k subcubes have identical known values, and is equal to one of the equations (4) otherwise. Initial conditions are given by 0-subcubes with values and probabilities given by the problem. The cost of the top node (the n-subcube) is just the cost of the optimal tree for the function; the tree itself can be recovered by walking down the lattice, choosing at each step to test the variable which gave rise to the least costly merging, until nodes with a known value (leaves) are reached.

Example 9: We shall make use of the function of Example 7 to show how the dynamic programming algorithm produces a decision tree with minimal expected testing cost. With 3 variables, the variables' space has 2^{sup3}=8 points, each identified by a unique combination of the 3 variables. The algorithm will build a lattice of 3^n nodes with $n \cdot 3^n$ pairs of edges, as shown in Figure 7. Since we are interested in the expected testing cost, we keep track at each node of the function's value, the node's probability, and the merging costs, where the cost of merging two $(i-1)$-subcubes is 0 if both subcubes have identical known values, and is equal to $c \cdot (p_1 + p_2) + c_1 + c_2$ otherwise, where c_1, p_1 (c_2, p_2) are the cost and probability of the first (second) $(i-1)$-subcube, respectively, and c is the testing cost of the variable used in merging. Since the cost of the least expensive merging which produces the top node is 5.05, that is the cost of the optimal tree for our problem. Large arrows in Figure 7 show which merging was least expensive at each step and allow the recovery of the optimal decision tree, in this case the linear testing sequence $x_1 x_2 x_3$ (the fifth tree in Figure 4). *box* Our sixth measure, the diagram storage cost, is not amenable to a solution by dynamic programming, since it supposes identification of common subtrees and thus a global (as opposed to dynamic programming's local) view of the subproblems. Thus, optimization of diagram storage cost is done by search techniques, principally branch-and-bound [ReiL67], a method which has also been applied to the optimization of the expected testing cost [ReiL66, Brei75b]. Recall that a branch-and-bound algorithm proceeds by always developing that partial solution which is potentially less expensive than any other (as determined by a lower bound function), often switching from one partial solution to another when lower bounds change, until one solution has been completely developed [Lawl66]. The lower bound function used for the diagram and tree storage costs [ReiL67] is based upon this simple fact: a non-redundant variable must appear at least once in any diagram. Thus, a rough lower bound can be derived by simply summing the storage costs of all the non-redundant variables. The lower bound used for the expected testing cost can be derived in an analogous fashion by considering the a priori probability that each

variable will be tested in any decision tree representation and modifying it to reflect the influence of the choice of a root [ReiL66, Brei75b, More80b], or it can be derived from first principles as a complexity measure on decision trees [More80a]. The principal disadvantage of the branch-and-bound method is that it may result in a near-exhaustive search of the possible trees and diagrams, a process that, in view of the dimension of the search space (as previously discussed), leads to intolerably long computations. In terms of algorithm analysis, the branch-and-bound technique is an exponential-time algorithm regardless of the input size. We now examine in more detail the bounding functions mentioned above and illustrate the use of branch-and-bound methods by a simple example. Again, the less mathematically inclined reader may wish to skip to the next section. A lower bound on the storage cost of a decision tree must incorporate the influence of the choice of a given variable to be of use in the branch-and-bound process. To achieve this end, the lower bound is modified by using a one-level "look-ahead"; if variable x_i is chosen for the root of the tree representing function f, then the lower bound is the sum of the storage cost of the chosen variable, x_i; and the storage cost of each x_j, $j \neq i$, times the number of restrictions, $f_{vbar\ x_i=k}$, for which x_j is non-redundant. For diagrams, the multiplicative factor in (ii) is modified to take into account the fact that x_j may play exactly the same role for some restrictions, that is, that for every combination of values of the remaining variables, either all the restrictions are equal or at most one is not constant. The lower bound on the expected testing cost of a decision tree can be derived from first principles by considering the development of a measure of the influence of a variable on the expected testing cost of decision tree representations. Any such measure should possess the following two properties: the measure is minimal (equal to zero) when the variable is redundant and maximal (equal to the variable's testing cost) when the variable is indispensable; and the measure is compatible with the tree structure, that is, if we denote such a measure for the variable x_i by $a_f(x_i)$, it must be the case that, for each $j \neq i$, $a_f(x_i) = \sum_{k=0}^{m_j-1} p(x_j = k) \cdot a_{f_{vbar\ x_j=k}}(x_i)$. More80a has shown that only one measure satisfies those two conditions: the activity of a variable, which is equal to the testing cost of the variable times the a priori probability that it will be tested (a concept related to the Boolean difference used in Boolean algebra [Thay81b]). The a priori probability that variable x_i will be tested is just the probability that, with all its other variables evaluated, the function still depends on x_i; this is easily computed in linear time. The lower bound used in [ReiL66, Brei75b] can then be defined as the sum of the testing cost of the root variable and the activities of the remaining variables [More80b].

Example 10: Consider again the Boolean function of our previous examples. The activities of the three variables are found to be $a_f(x_1) = 1 \cdot prob(x_2 = 1 \text{ and } x_3 = 0) = 0.3$, $a_f(x_2) = 2 \cdot prob(x_3 = 1 \text{ or } x_1 = 0) = 1.4$, $a_f(x_3) = 6 \cdot prob(x_1 = 1 \text{ and } x_2 = 1) = 2.4$. Now the lower bound on the expected testing cost of a decision tree for f with root x_i, $lb_f(x_i)$ can be computed for each variable $lb_f(x_1) = t(x_1) + a_f(x_2) + a_f(x_3) = 4.8$, $lb_f(x_2) = t(x_2) + a_f(x_1) + a_f(x_3) = 4.7$, $lb_f(x_3) = t(x_3) + a_f(x_1) + a_f(x_2) = 7.7$. Thus the branch-and-bound algorithm chooses to develop the tree rooted in x_2. The left subtree is then a leaf labelled 0 so that only two possible

partial trees arise, depending on the choice of the variable tested at the root of the right subtree. Lower bounds are in turn computed for these. We now have four partial subtrees, pictured with their lower bounds in Figure 8a. The algorithm will choose to develop the first partial tree (rooted in x_1), since it is now the least expensive; this yields four partial trees for a total of seven partial trees, pictured with their lower bounds in Figure 8b. Now the algorithm will return to the fifth tree (rooted in x_2) and complete it; since its final cost, 5.05, is lower than the bound on any partial tree, the completed tree is optimal. *box* In the context of logic--particularly Boolean--functions, specialized methods have been devised which attempt to use some of the standard tools of logic (such as reduction to canonical or minimal formulae, decompositions, etc.) in order to construct decision trees and diagrams with minimal storage or expected testing cost [Mich78, Thay78, Cern79a, Thay81a]. The minimization of storage or expected testing cost for decision trees has been approached for Boolean [Cern79a] and multivalued [Thay78] logic functions by considering a special class of subfunctions, which the authors call **T-terms**. Starting with the terms of order 0, which are just logic cubes (in the sense of switching theory [Harr65]), terms of successively higher order are constructed by consensus operations. (That is, for each variable, x_i, one takes the consensus of a T-term of order n and one of order $k \leq n$ with respect to x_i; the result is a T-term of order $n+1$ if it is not already a T-term of lower order and if it is independent of x_i.) Only the prime terms are kept in the construction process (where a prime term is a term not contained in any other term of the same order). A simple procedure is then used to derive a tree optimal with respect to worst-case testing cost, expected testing cost, or storage cost. Although the algorithm sheds light on the relationship between Boolean formulae and optimal decision trees, it is not of practical interest (except for minimizing diagram storage cost) since it contains a hard problem: the prime terms of order 0 are just the prime implicants of the function [Harr65] and obtaining them, even from complete function descriptions, is known to be NP-hard [Mase80]. As a result, the procedures proposed may require exponential time under any input size. A similar drawback is present in the algorithm published by Mich78, which converts extended-entry decision tables to decision trees with minimal storage or expected testing cost, since the algorithm starts by establishing a minimal disjoint set cover for the function, a process known to be NP-hard [Gare79]. An extension of T-terms was recently proposed by Thay81a for the design of decision trees and diagrams with minimal storage cost. This formulation is based on a class of functions called P-functions, where a P-function for the Boolean function f is a pair, $<g, h>$, of functions such that f and h are equal when restricted to the points where g evaluates to 1. A composition operation is defined which allows the building of a lattice of P-functions, from the lowest order (with $h = 1$ or $h = 0$) to the highest order (with $g = 1$). Again, only prime P-functions are retained in building the lattice. A search procedure generates optimal decision diagrams from the lattice of prime P-functions. The generation of optimal decision trees, however, requires that prime P-functions be replaced by prime P-cubes, i.e., prime P-functions restricted so that they are logic cubes. Since the lowest order P-cubes are comprised of the prime implicants of the function and its complement, we find anew the NP-hard subproblem, so that the synthesis of optimal decision trees by P-functions requires exponen-

tial time and is therefore of little practical interest. (It must be noted that we do not imply that finding prime implicants is an unsurmountable task; in fact, several algorithms for that purpose have been studied and shown to do well in practice [Slag70, Hulm75]. Our point is that the methods described above, which incorporate this NP-hard problem as only a small part of the complete work, cannot compare with the dynamic programming method.) On the other hand, the construction of optimal diagrams, while exponential (no analysis is provided with the algorithm, but the generation of all prime P-functions may clearly require that much time), remains of interest since no substantially better solution is known. We now proceed to a closer examination of the composition of P-functions, followed by an example. Once again, the less mathematically inclined reader may wish to skip to the next section. If f has n variables, it has up to $2^{2^n}-1$ P-functions on which a lattice structure can be established, from the lowest to the highest order by means of the following non-commutative composition law (designed to be compatible with Shannon's decomposition). A P-function of order $k+1$, $<g, h>$, is obtained from two P-functions of lower order (one of order k and the other of order no larger than k), $<g_0, h_0>$ and $<g_1, h_1>$, by using, for each x_i, the formula: $< g, h > = < g_{0 vbar\ x_i=0} \cdot g_{1 vbar\ x_i=1},\ \overline{x_i} \cdot h_0 + x_i \cdot h_1 >$, which we denote $< g_0, h_0 > cidot_i < g_1, h_1 >$. In a sense, the resulting P-function of order $k+1$ corresponds to our state of knowledge prior to testing variable x_i. Indeed, substituting $x_i = 0$ into $<g, h>$ yields $< g_{0 vbar\ x_i=0} \cdot g_{1 vbar\ x_i=1},\ h_0 >$, and substituting $x_i = 1$ yields $< g_{0 vbar\ x_i=0} \cdot g_{1 vbar\ x_i=1},\ h_1 >$, which shows how the second function in the pair reflects our increased knowledge about the function f (until that second function is a constant, meaning that the evaluation of f is complete), while the first function provides us with information about the path of evaluation followed so far.

Example 11: Consider the Boolean function of our previous examples. The two prime P-functions of order 0 are: $A_0 = < f, 1 >$ and $A_1 = < f, 0 >$. From those two functions, we can form six P-functions of order 1, three of which are prime: $B_0 = A_0 cidot_1 A_1 = < x_2\overline{x_3}, \overline{x_1} >$, $B_1 = A_1 cidot_2 A_0 = < \overline{x_1} + x_3, x_2 >$, $B_2 = A_1 cidot_3 A_0 = < x_1 x_2, x_3 >$. Using now the five prime P-functions of order 0 and 1, we can form 54 P-functions of order 2 (not all distinct), four of which are prime: $C_0 = A_0 cidot_1 B_2 = B_0 cidot_3 A_0 = < x_2, \overline{x_1} + x_3 >$, $C_1 = A_1 cidot_2 B_1 = B_1 cidot_2 A_0 = < \overline{x_1} + x_3, x_2 >$, $C_2 = A_1 cidot_3 B_1 = < x_1 + \overline{x_2}, x_2 x_3 >$, $C_3 = B_1 cidot_1 A_1 = < \overline{x_2} + \overline{x_3}, \overline{x_1} x_2 >$. Finally, we can use our nine prime P-functions of orders 0, 1, and 2 in order to obtain the single prime P-function of order 3 at the top of the lattice, $D_0 = <1, f >$: $D_0 = A_1 cidot_2 C_0 = C_2 cidot_2 C_0 = C_3 cidot_2 C_0 = C_1 cidot_1 C_2 = B_1 cidot_1 C_2 = C_3 cidot_3 C_1 = C_3 cidot_3 B_1$. The corresponding lattice is shown in Figure 9, where a number, i, in a circle has been used to denote a composition with respect to the i-th variable. A search through the lattice shows that the optimal diagrams have three nodes for a storage cost of 6; the diagram of Figure 2 was one of those.*box* Since decision tree optimization is an NP-hard problem under polynomial-size inputs [More80b], it is necessary to develop some heuristics that will allow the fast construction of good, albeit not optimal, solutions. Indeed, some such heuristics have been proposed even before an optimal algorithm was developed: it appears that the so-called "splitting" heuristic, to be discussed below, was known

to Aristoteles and Theophrastus [Voss52], and many heuristics were proposed for the conversion of decision tables [Mont62, Egle63, Poll65] before the publication of the branch-and-bound solution of ReiL66. All published heuristics are of the so-called greedy type, that is, they perform a local, step-by-step optimization. Three main types of criteria are used; the apparently large variety is due to attempts at accommodating tests with variable outcomes (as in most biological applications [Brow77, Gowe75, PayR81]), or to minor differences in preprocessing (such as attempts to put decision tables in a canonical form [Shwa75]) or in the extent of look-ahead used (while most strategies use no look-ahead at all, Seth80 has suggested a one-step look-ahead and Mich78 proposed a range of look-aheads, from 0- to k-step). These three criteria will be discussed in some detail below. (Since the discussion of the first--and most important--of these criteria involves some mathematical manipulations, we have once again organized its discussion in two parts, grouping all mathematical concepts in the second.) In this strategy, commonly used for the (near) minimization of the expected testing cost, the problem is viewed as one of refining an initial uncertainty about the function's value into a certitude. At each step, the test of a variable diminishes the universe of possibilities, thereby removing a certain amount of ambiguity. In information-theoretic terms, the initial ambiguity of a (partial) function is expressed by the entropy of the function (for a lucid exposition on the topic, see the original paper of Shan48). The ambiguity remaining after testing a variable can be computed as the average ambiguity among the restrictions. This allows the computation of the ambiguity removed (or, equivalently, the information gained) by testing that variable. The information heuristic then chooses at each step that variable which removes the most ambiguity per unit testing cost. The previously mentioned splitting heuristic is a special case of the information heuristic for identification problems. It is well-known [Shan48] that the removed ambiguity is maximized by letting all restrictions have equal probability; thus, when all variable costs are unity, the information heuristic chooses that variable which "splits" the set of objects into subsets with most nearly equal probabilities. In the case of binary identification problems with equally likely objects and unity costs, this means selecting that test which splits the objects into subsets of most nearly equal size. Yet another aspect of the splitting heuristic is a criterion used for binary identification problems [Gyll63, Chan65] which selects that variable which separates the largest number of pairs of values : if n is the total number of values and k the number of values put into one subset, then $k \cdot (n-k)$ pairs are split, which expression is maximum for $k = n/2$. (Choosing that variable which separates the largest number of pairs could be described as the **separation** heuristic; this criterion also appears in a variety of forms, some of which attempt to include tests with variable outcomes [Brow77, PayR80].) Descriptions of the heuristic and its variants abound [Klet60, Resc61, Osbo63, Mand64, Gowe71, Gana73, Shwa74], but it was not until later that the performance of the information heuristic was analyzed. Gare74 studied its application to identification problems and showed that, although it is quasi-optimal when all possible tests are available (a result dating from Zimm59), there are problems for which it can construct trees with an expected testing cost arbitrarily larger than the optimal. This result disproved a longstanding conjecture that the splitting algorithm was optimal for identification problems with

equally likely objects [Klet60, Osbo63]; such a result is, of course, predictable now in view of the NP-hardness of the problem since an optimal polynomial algorithm would disprove the widely held opinion that NP-hard problems actually require exponential solutions. However, Hung74 proved that the heuristic is, on the average, asymptotically optimal; that is to say, the ratio of the average cost of trees built with the information heuristic to that of the optimal trees converges to 1 as the problems get larger. This result must be qualified by the observation that most functions have an expected testing cost fairly close to maximum, so that the asymptotic ratio used in Hung74 is in general fairly small for any heuristic. Indeed, More81b showed that completely specified Boolean functions of n variables with unity costs and uniform probability distribution have an asymptotic expected testing cost of $n-1$, so that in this case the asymptotic average ratio must be 1 for any heuristic. Recently Hart82 presented a generalization of the information heuristic which allows for more complex objective functions (including the acquisition cost) and offers a trade-off between the complexity of the construction and the upper bounds that can be placed on the size of the resulting solution. The information heuristic is efficient: for an input size of $O(S)$, it takes time $O(S \cdot \log\ S)$ which, while barely faster than the optimal dynamic programming solution for exponential size inputs, compares very favorably indeed with the exponential-time algorithms used for identification problems. We now examine the entropy computations in some detail; once again--and for the last time--the less mathematically inclined reader may wish to skip to the next section. The expression for the entropy of a function, f, is [Shan48]: $H_f = -\sum_v p(f=v) \cdot \log_2 p(f=v)$, where $p(f=v)$ is the probability that f takes the value v and the sum is taken over all the values v in the range of f (values of $p(f=v)$ are normalized so that their sum over the values of v is equal to 1). After testing variable x_i, the remaining ambiguity, $H_f(x_i)$, is the average ambiguity among the restrictions: $H_f(x_i) = \sum_{j=0}^{m_i-1} \left(p(x_i=j) \cdot H_{f_{vbar\ x_i=j}} \right)$ Hence the ambiguity removed by testing x_i is the quantity: $I_f(x_i) = H_f - H_f(x_i)$. This quantity is computed for each variable; the information heuristic then chooses at each step that variable which has the greatest ratio, $I_f(x_i)/t_i$. For identification problems, the expression for the removed ambiguity can be simplified to: $I_f(x_i) = -\sum_{j=0}^{m_i-1} p(x_i=j) \cdot \log_2 p(x_i=j)$, which is seen to be of the same form as (7). Thus maximizing the removed ambiguity in an identification problem involves finding that variable x_i such that the probabilities $p(x_i=j)$ are most nearly equal to each other.

Example 12: Let us use once more the function of Example 7. The entropy of the function is: $H_f = -[p(f=0) \cdot \log_2 p(f=0) + p(f=1) \cdot \log_2 p(f=1)]$

$= -[0.6 \cdot \log_2 0.6 + 0.4 \cdot \log_2 0.4] = wig\ 0.971.$ The ambiguity remaining after testing x_1 is: $H_f(x_1) = p(x_1=0) \cdot H_{f_{vbar\ x_1=0}} + p(x_1=1) \cdot H_{f_{vbar\ x_1=1}} = wig\ 0.5 \cdot 1 + 0.5 \cdot 0.881 = wig\ 0.941.$ Thus the information gain of x_1 per unit testing cost is: $I_f(x_1)/t(x_1) = wig\ (0.971 - 0.941)/1 = 0.03$. Similarly, we find: $I_f(x_2)/t(x_2) = wig\ (0.971 - 0.625)/2 = 0.173,\ I_f(x_3)/t(x_3) = wig\ (0.971 - 0.652)/6 = wig\ 0.053,$ so that the

heuristic will choose to test x_2 first. Since the restriction for $x_2 = 0$ is constant, only the restriction for $x_2 = 1$ remains. The information gains per unit cost of x_1 and x_2 for that restriction are: $I_{f_{vbar\ x_2=1}}(x_1)/t(x_1) = wig\ (0.961 - 0.382)/1 = 0.579,\ I_{f_{vbar\ x_2=1}}(x_3)/t(x_3) = wig\ (0.961 - 0.195)/6 = wig\ 0.128,$ so that x_1 is tested next. The completed tree is in this case the optimal tree developed previously. *box* A class of simple heuristics can be obtained for any problem by using the bounding functions of branch-and-bound algorithms and doing local optimization on their basis, in effect using branch-and-bound without backtracking. This approach is of no particular interest for the minimization of storage cost (although it has been used for that purpose [Rabi71, Yasu71]) since the existing bounds are too loose; it is, however, applicable to the minimization of the expected testing cost, since the lower bound based on activity is in general tighter. The activity heuristic has the same computational requirements as the information heuristic. More80b provided an analysis of its performance, showing that the worst-case ratio for completely specified Boolean functions with unity costs and uniform probabilities is limited to 2, but can be arbitrarily large if non-uniform probabilities are allowed. This heuristic appears to be of less interest than the information heuristic, since it performs best for "dense" problems, that is, those where the function is specified on most of its domain, which are precisely those problems that can be efficiently solved by the optimal dynamic programming algorithm. Finally, a similar approach has been taken by some authors for biological identification problems, using rough lower and upper bounds on the number of tests needed to complete an identification (see the thorough studies of Brow77 and PayR81). In one such approach, it is postulated that the subtree will completed optimally (that is, following Huffman's procedure--even though only a small proportion of all tests is available); the resulting lower bound is used for deriving a selection criterion [Dall74, Brow77]. Alternatively, it is assumed that the tree will be completed by a linear sequence of simple tests; the resulting estimate is an upper bound under most conditions and allows the derivation of another selection criterion [PayR81]. However, those criteria are based on rather simplistic bounds and thus susceptible to large errors; despite the lack of either theoretical or practical results about their performance, one can safely predict that their average performance is worse than that of the other criteria examined so far. Since Boolean functions are conveniently expressed by formulae, special heuristics can be developed which are based on characteristics of the formulae such as number of terms or literals. One such heuristic, proposed by Halp74 for the minimization of the expected testing cost, necessitates the generation of all prime implicants for the function and its dual: the implicants are then ranked in terms of their probability to cost* The cost of an implicant is that of the optimal tree for it; that tree is easily constructed [Ries63, Slag64] since tree representations of conjunctions of variables are just linear test sequences. ratio. Variables which appear in both the best implicant for the function and that for its dual are then selected (at least one such variable must exist). Halp74 proved that this strategy is optimal for symmetric functions (those that remain invariant under any permutation of the variables), but offers no analysis of performance in the general case. Brei75a presented a similar heuristic for monotone Boolean functions with unity costs and uniform probability distribution,

which uses the minimal disjunctive form of the function (this form is unique for monotone functions [Harr65]); in a later analysis [Brei78], it was shown that trees constructed by this rule can have an expected number of tests at least (n/log n) times larger than the optimal trees for functions of n variables. Both heuristics apply only to completely specified Boolean functions and require exponential time since the generation of all prime implicants and/or the minimization of the disjunctive form are NP-hard problems [Mase80]. Since dynamic programming offers an $O(S^{\log_2 3} \cdot \log S)$ optimal solution to the same problem, these heuristics are of interest only when the function is already specified by its prime implicants or its minimal disjunctive form. In this section, we describe the main fields of application and review related results. We distinguish four fields: decision table programming; diagnosis, identification, and pattern recognition; logic and program design; and analysis of algorithms. Of these, only the last three are treated, since decision table programming is chiefly concerned with the construction of optimal decision trees, a topic with which we dealt in the previous section. Gare72a argues that most identification problems of human origin include a large number of simple tests (of the type "is the unknown object of type i ?") plus a smaller number of "well-splitting" tests. Indeed, this is how most of us approach the typical identification problem of "twenty questions", starting with general, well-splitting questions (e.g., "is it mineral") and ending the game with simple questions (e.g., "is it an aardvark"). Several large identification problems approximate this description, notably in botanical and biological classification [Moll62, Pank70, Mors71, Will80]. Gare72a has described a dynamic programming algorithm especially designed for this type of problems which constructs identification trees with minimum expected testing cost. In most identification problems, many more tests are present than are needed for the identification of all objects. In consequence, several researchers have studied the problem of obtaining a minimal set of tests; we recognize in this problem the minimization of the total acquisition cost. Such an optimization is important when individual tests are time-consuming and promptness in identification essential (as in medical diagnosis [PayR80, Will80]), so that parallel, rather than sequential, testing is used. Unfortunately, the minimum test set problem, as it is known, is itself NP-hard [Gare79]. As a result, splitting heuristics based on the number of split pairs have been proposed for the construction of suboptimal solutions [Gyll63, Chan65]; however, no performance analysis was supplied. (The general analysis of the splitting algorithm presented in the previous section does not apply here since the goals are quite distinct.) Some preliminary analytical results as well as extensive experimental data can be found in More82. This problem is closely related to that of finding prime implicants for functions of Boolean variables, since each minimal set of prime implicants determines an irredundant set of tests for the problem. Hence methods have been proposed that first find all prime implicants and then attempt to find a set of prime implicants which minimizes the number of distinct variables used (see the review of PayR80, pp.261-263). Since decision trees model multistage branching decision processes, they find a particularly important application in sequential, or more precisely, hierarchical, pattern recognition (see Kana79 for some general considerations about the advantages of hierarchical approaches). In the simplest case, a pattern recognition problem is deterministic and reduces to an identification prob-

lem. In general, however, a type (called a **class**) of objects is not absolutely characterized by selected combinations of test values (called **features**); rather, the problem is of a statistical nature, such that each combination of features is distributed among all the classes. (This model of probabilistic identification also corresponds to the fuzzy decision tables discussed in Kand80.) If the set of features has sufficient power of discrimination, the probability distribution of each combination of features will exhibit one strong peak for some class, so that an object possessing this combination of features can be classified in that class with a low probability of error. At times, however, it may be advantageous to trade accuracy for speed and allow an object to be classified in a patently wrong class in order to gain on response time. As a consequence of this additional freedom, decision trees for pattern recognition purposes are subject to yet another optimization criterion: minimum overall probability of misclassification.

Example 13: Consider the following simplified pattern recognition problem with three classes, C_1, C_2, C_3, and three binary features, x_1, x_2, x_3. The testing costs of the three variables (features) are: t: x_1 *rarrow* 1 x_2 *rarrow* 2 x_3 *rarrow* 3, and the probability distribution on the variables' space is given by:

p_v: $(0,0,0)$ *rarrow* 0.10 $(1,0,0)$ *rarrow* 0.20

$(0,0,1)$ *rarrow* 0.20 $(1,0,1)$ *rarrow* 0.10

$(0,1,0)$ *rarrow* 0.10 $(1,1,0)$ *rarrow* 0.10

$(0,1,1)$ *rarrow* 0.05 $(1,1,1)$ *rarrow* 0.15

The distribution of each combination of features among the classes is given below, where the probability that a given combination corresponds to class i is given by the i-th value of the triple.

p_c: $(0,0,0)$ *rarrow* $(0.10, 0.85, 0.05)$

$(0,0,1)$ *rarrow* $(0.01, 0.98, 0.01)$

$(0,1,0)$ *rarrow* $(0.20, 0.10, 0.70)$

$(0,1,1)$ *rarrow* $(0.80, 0.10, 0.10)$

$(1,0,0)$ *rarrow* $(0.80, 0.10, 0.10)$

$(1,0,1)$ *rarrow* $(0.90, 0.05, 0.05)$

$(1,1,0)$ *rarrow* $(0.05, 0.05, 0.90)$

$(1,1,1)$ *rarrow* $(0.10, 0.00. 0.90)$

The two distributions allow us to compute the a priori probability of each class, i.e., the probability that a unknown object belongs to that class: $p(C_1) = 0.342$ $p(C_2) = 0.326$ $p(C_3) = 0.332$ Since each combination of features must be classified in some class, the strategy which minimizes the probability of misclassification is obviously to classify a combination of features in the class for which it shows the largest probability; thus we get the assignment:

f: $(0,0,0)$ *rarrow* C_2 $(1,0,0)$ *rarrow* C_1

$(0,0,1)$ *rarrow* C_2 $(1,0,1)$ *rarrow* C_1

$(0,1,0)$ *rarrow* C_3 $(1,1,0)$ *rarrow* C_3

$(0,1,1)$ *rarrow* C_1 $(1,1,1)$ *rarrow* C_3

Hence the probability of misclassification of an object with the combination of features (0,0,0) is 0.10+0.05 = 0.15; since that combination of features occurs with a probability of 0.10, it contributes a total of $0.10 \cdot 0.15 = 0.015$ to the overall minimal probability of misclassification; working similarly with the other combinations of features, the latter probability is found to be: $p_{e_{\min}} = 0.134$. A decision tree with that overall probability of misclassification is shown in Figure 10a together with the probability of its leaves; this tree has an expected testing cost of 4. A different tree is pictured in Figure 10b; this tree classifies the combination (0,1,1) in class C_3--clearly not an optimal choice in terms of classification accuracy. However, this tree has an expected testing cost of only 2.6 and its overall probability of misclassification is found to be 0.164, barely larger than optimal. Thus it is a difficult task to decide which tree is best; other criteria must be used, such as penalties due to misclassification, maximum permissible response time, etc. *box* Faced with such a variety of design criteria, researchers in the field have explored different routes. A tree is often synthesized directly from the problem without attempt to optimize its testing cost, but using heuristics designed to minimize the probability of misclassification [You76, Roun79]. Haus75 and Wu75 described a local optimization algorithm, based on measures of interclass separation, for the semi-automatic design of a decision tree from a known set of features. When the features are known in advance, a procedure based on dynamic programming due to Datt81 can be used to build a decision tree with minimal cost, where the cost criterion includes the expected testing cost and the cost of misclassification; a similar approach based on game theory was described in Slag71 and a third in Kulk76. When the class assignments are made (and the probability of misclassification therefore fixed), the pattern recognition problem reduces to the description of a (partial) function; Bell78 models this case by decision tables and discusses their conversion to sequential testing procedures (although the optimal algorithm of Baye73, PayH77 is not mentioned). The accuracy of decision tree classifiers depends upon such considerations as sample statistics (e.g., sample size) and the number and intercorrelation of features; Kulk78 studied the problem under simplified assumptions and concluded that hierarchical classifiers, as their single-stage counterparts, may suffer from the "dimensionality" problem, that is, show decreasing performance if the number of features is increased beyond a certain threshold (which depends on the sample size). We have already mentioned that decision trees and diagrams for Boolean functions naturally give rise to multiplexer implementations. Such implementations are attractive since the resulting circuits have few interconnections and lend themselves well to large-scale integration (for instance, binary trees form an efficient interconnection pattern [Horo81]). Moreover, multiplexer networks can be used as universal logic modules (ULMs) [Tabl76, Voit77], thereby reducing the number of basic components needed for logic design.

Decision tree representations of Boolean functions exhibit several advantages over Boolean formulae. Lee59 showed that at most $O(2^n/n)$ diagram nodes are required to represent any Boolean function of n variables, which compares very favorably with the $O(2^n/\log\ n)$ operators that may be needed by an unfactored Boolean formula [Sava76]. Moreover, every operator of the Boolean formula must be carried out in order to evaluate the formula, so that up to $O(2^n/\log\ n)$ operations may be performed, while a decision diagram will never require more than n variable evaluations. Thus decision diagrams express Boolean functions at least as compactly as Boolean formulae and are greatly more efficient as an evaluation tool. (The latter property is used, e.g., for the repeated evaluation of Boolean queries in a large data base [Wong76].) Finally, decision diagrams lend themselves to composition and recursion, as will be shown in the next section, while the same operations are very difficult to carry out using formulae. Some of those advantages were rediscovered and some pointed out for the first time by Aker78, who investigated the use of binary decision diagrams (with a slightly different definition) in testing digital systems. In conjunction with the evaluation of Boolean functions, it must be noted that almost all Boolean functions have a pessimal worst-case testing cost (that is, all variables are tested on at least one path). This result, due to Rive76a, was later complemented by More81b, who proved that all symmetric and all linearly separable (also called threshold) Boolean functions possess this property. An important consequence of these results is that synchronous multiplexer implementations of Boolean functions cannot, in most cases, be optimized with respect to their propagation delay, since this delay is determined by the longest path through the network. Finally decision diagrams can be used to model the control structure of a program (as advocated in Prat78, where they are called "atomic digraphs"), in particular in relation to Ianov's schemata [Iano60]. It then becomes important to recognize identical structures, that is, to decide whether or not two decision diagrams are equivalent. This problem is known to be NP-hard [Fort78]; however, Blum80 described an efficient algorithm which solves the equivalence problem probabilistically (that is, which provides an answer correct within a given--and refinable--percentage of error). The worst-case number of tests (the height) of a decision tree indicates a minimum number of argument evaluations which must be performed in order to compute a function. As such, finding the minimum height of any tree representation of a function is a useful technique for deriving bounds to be used in the worst-case analysis of algorithms. The above-mentioned results of Rive76a and More81b, showing that large classes of Boolean functions require any decision tree representation to have maximal height, are an example of such analysis; indeed, Rive76b built upon these results to prove some lower bounds on the complexity of graph algorithms based upon a matrix representation. The worst-case number of evaluations must be at least equal to the ratio of the initial ambiguity of the function to the upper bound on the information supplied by each evaluation. Since the evaluation of a k-ary variable provides at most $\log_2 k$ bits of information, the height of any tree for a function, f, of k-ary variables, must obey the relation: $h_{\min} \geq H_f/log_2 k$. This relation has been used to provide lower bounds on the complexity of several combinatorial problems, such as sorting [Knut71] and various set operations [ReiE72]. The decision tree approach has recently been generalized to handle probabilistic, non-deterministic, and alternating

models of complexity [Manb82]. The decision diagram model of representation as described so far is limited to (partial) functions. Several extensions have recently been proposed, to include composition of diagrams [Aker78, More80b] and modelling of relations and simple recursion [More80b]. Whereas a function assigns at most one value to each combination of variables, a relation may assign any number of values, i.e., any subset of the set of values. Thus, in particular, a relation accurately models an ambiguous decision table, where the same rule (the same combination of variables) may specify more than one set of actions. In accordance with King73, we assume that, when inconsistent rules apply, any of the assigned action sets may actually be chosen for execution. In terms of relations, a decision tree implementation can choose to specify for each combination of variables any (or some, or all) of the values from the subset assigned to that combination by the relation. Since combinations of variables for which the function (relation) is not defined are usually allowed to take any convenient value, we may assume that such combinations are in fact related to the whole set of values. Another consequence of our assumptions is that a relation is constant exactly when the intersection of the subsets of values assigned to all the combinations of its variables is not empty. (For one can choose to use any of the values present in the intersection for each of the combinations of variables, thereby effectively transforming the relation into a constant function.) Under such assumptions, all of the results discussed so far apply to the representation and evaluation of relations [More80b]. A particularly important tool in the analysis of problems is decomposition, which tries to simplify a problem by partitioning it into smaller parts (the rationale being that the complexity of the whole is more than the sum of the complexities of its parts); "divide-and-conquer" is a time-honored aspect of decomposition. Conversely, composition is an important tool in synthesis. We have seen that decision trees induce a natural decomposition: Shannon's decomposition; thus it remains to demonstrate how to compose decision trees and diagrams. Two such compositions can be distinguished: leaf composition and node composition. Leaf composition stems from the simple hierarchical idea of a "tree of trees". As an example, consider a pattern recognition problem such as bird identification. To most of us, identifying a bird as a "sparrow" or a "dove" is sufficient; to a bird-watcher or an ornithologist, however, this is but a vague classification which must be greatly refined to include species and subspecies. This suggests a two-step identification, in which a rough classification is first made, followed by a highly specialized procedure for further refinements. This offers several advantages: it is practical since it can be of use to both uninitiated and specialists; it is efficient since the evaluation can be profitably (for the uninitiated) stopped after the first stage; and it can be greatly optimized in the second stage since it is then possible to design highly specialized tests. The first step in such a design consists of a single decision tree; most of the leaves of the tree, however, do not give a value, but rather designate another decision tree to be used in the second step. Thus, one can stop the evaluation upon reaching a leaf of the first tree, taking the "name" of the second tree as the result of the evaluation, or continue evaluation by proceeding to the second tree. The latter choice results in the composition of the two trees, called leaf composition since it replaces a leaf by another tree. Formally, then, leaf composition of two trees is the process of attaching the second tree in place of appropri-

ate leaves in the first tree. (The analog in the software world is a transfer of control between modules without transfer of information, such as chaining.) Clearly, the second tree cannot share variables with the first lest the composed tree test the same variable twice on some path. A special case of interest is the composition of decision trees for Boolean functions where the second tree is attached in place of every leaf with the same label in the first tree. One easily verifies that such a composition results in a logical OR of the two functions when the leaves labelled "0" are replaced, and in a logical AND when the leaves labelled "1" are used instead. Moreover, the composition is then commutative (in terms of the function it yields), just as the logical operation that it implements. Figure 11a shows two trees which are OR-composed in Figure 11b and AND-composed in Figure 11c. The behavior of the various optimization criteria under leaf composition is simple. The storage cost of the composition is the sum of the cost of the first tree and, for each replaced leaf, of the cost of the second tree. This can be simplified for diagrams; since only one leaf of each label can exist, the storage cost of the composed diagram is just the sum of the costs of the component diagrams. The expected testing cost of the composition is the sum of the cost of the first tree and of the cost of the second tree, the latter multiplied by the probability that one of the replaced leaves will be reached. Thus all the techniques used for the optimization of decision trees are applicable to the optimization of composed trees (see More80b). In connection with the composition of Boolean decision trees mentioned above, Perl76 proved that, on the average, the order of composition is irrelevant to the expected testing cost of the composed tree. (Notice, however, that this is clearly not the case for the OR-compositions illustrated in Figure 11b.) Whereas the rationale behind the leaf composition was the progressive refinement of function values, node composition introduces a refinement of the values of the variables. Thus the former is associated with an explicit tree hierarchy, while the latter induces a functional hierarchy. In a node composition, a k-ary variable is replaced by a tree with at least one leaf labelled with each value from 0 to $k-1$; this tree specifies how to evaluate the variable it replaces. (Thus the software analog is the use of an auxiliary procedure which returns a value, that is, a subroutine call.) The effect of node composition on functions expressed by formulae is just the substitution in the first function's formula of the second function's formula for each appearance of the node variable. In terms of trees, the composed tree is obtained by using the following procedure for each occurrence of the specified node: replace the node by the second tree and attach the j-th subtree of the node to each leaf labelled j in the second tree. As for leaf composition, the trees used in node composition must not share any variable. Figure 12 illustrates a node composition. Aker78 proposed this composition and studied its use in the design and analysis of logic functions. It is noted that the behavior of the optimization criteria under node composition is somewhat complex. While the diagram storage cost of a composition is simply the sum of the cost of the first diagram and, for each replaced node, of the cost of the second diagram, the other costs depend on exact structure of the second tree. In particular, it is necessary to know how many leaves of the second tree share the same label, say label j, since the j-th subtree of the replaced node will be attached to each of these leaves. However, the optimization methods described in this article can be applied with suitable modifications. Both modes of

composition can be used at once. Recent research on the problem of bacteriological identification in a clinical environment [Shap81] suggests as the initial model a user-specified identification tree which simply describes a hierarchy of classes and subclasses of bacteria (as determined by local factors such as common mode of treatment in initial stages, likelihood of occurrence in the geographical area, etc.). The hierarchy involves leaf composition while the actual tests to be used for identification are specified by node composition. In discussing both modes of composition, we have purposefully avoided a delicate problem: what if a tree is composed with itself? In such a case, the leaves or nodes of the first tree are replaced by the identical tree, so that more replacements are possible, and so on. This creates an infinite recursion, giving rise to an infinite tree (or a diagram with cycles). Although composing a tree with itself may appear contrived, recursion is a sufficiently fundamental phenomenon that a study of its effects on decision trees is warranted. Unfortunately, very little work has been done in this area. More80b studied the recursion due to a single leaf composition and showed that the concepts of expected testing cost, diagram storage cost, and activity can be extended to such simple recursive trees. Although the tree storage cost is clearly infinite, as is the worst-case testing cost, the expected testing cost is generally finite. This stems from the fact that there usually is a non-zero probability, call it e, of reaching a non-replaceable leaf in the component tree, so that the probability of recursing one level, $1-e$, is less than one. Now, the probability of recursing k levels is just $(1-e)^k$, so that the average number of recursion levels used before termination is $r_{av} = \sum_{k=0}^{infinity} (1-e)^k = \frac{1}{e}$. This factor can be used for transforming the expected testing cost of a single, non-recursive copy of the tree into the expected testing cost of the recursive tree; the activities of the variables can be similarly computed. Decision trees have long been used for purposes of fault diagnosis, as previously seen. Such uses, however, apply decision trees to the analysis of a discrete function which is not that which they represent. Aker78,79 proposed that binary decision diagrams be used as the basis for developing tests for the Boolean function which they represent. Some of the reasons given have been discussed above, such as compactness, good structure, and existence of composition. Other reasons include the close relationship between binary decision diagrams and standard logic design and the ease of automation in handling diagrams. More81a advocated the use of decision trees as alternate models for system analysis. A standard tool in the analysis of system reliability is the fault tree [Barl75], which is just a graphical representation (using AND and OR gates) of the Boolean function describing a system's state in terms of the state of its components (where the only possible values are "working" and "failing"). Fault trees are used for the assessment of the overall reliability of a system and of its sensitivity to the state of various components. The same analysis can be performed using decision trees and activities, with added advantages: (i) as seen, decision trees are more efficient than formulae (which is essentially what fault trees are); (ii) decision trees can be used for multivalued functions, whereas fault trees are restricted to Boolean functions; and (iii) decision trees can include recursion and general composition operations, while fault trees are limited to an equivalent of node composition. A possible drawback, however, is that the manipulation of Boolean formulae, while inherently inefficient, is well understood and has been successfully implemented

(e.g., see Worr75), whereas that of decision trees is still in its infancy. This article has provided a unified framework of definitions and notation for decision trees and diagrams; it has examined the problem of optimization, reviewed the main applications and contributions, and described some recent developments. While often difficult to optimize, decision trees and diagrams emerge as efficient representations of discrete functions, of particular interest in pattern recognition, logic design, programming methodology, and system analysis. Although several research programs are actively concerned with decision trees, further areas of study have been identified, notably the quality of optimization heuristics and the use of general recursion. The author wishes to thank Professors R. C. Gonzalez and M. G. Thomason at the University of Tennessee for their advice and encouragement, as well as for introducing the author to decision trees and collaborating on his research; Professor H. D. Shapiro at the University of New Mexico, for many fruitful discussions and a very careful review of the manuscript; and the editor and referees, for detailed and constructive comments. Akers, S. B. "Binary decision diagrams," IEEE Trans. Comp. **TC-27**, 6 (1978), 509-516. Akers, S. B. "Probabilistic techniques for test generation from functional descriptions," in Proc. 9th IEEE Fault-Tolerant Comp. Symp., 1979, 113-116. Barlow, R. E., Fussell, J. B., and Singpurwalla, N. D., Eds. Reliability and Fault Tree Analysis, SIAM Press, Philadelphia, PA., 1975. Bayes, A. J. "A dynamic programming algorithm to optimize decision table code," Austral. Comp. J. **5**, 2 (1973), 77-79. Bell, D. A. "Decision trees, tables, and lattices," in Pattern Recognition: Ideas in Practice, Batchelor, B. G., Ed., Plenum Press, New York, 1978, Chap. 5. Blum, M., Chandra, A. K., and Wegman, M. N. "Equivalence of free Boolean graphs can be decided probabilistically in polynomial time," Inf. Proc. Letters 10, 2 (1980), 80-82. Breitbart, Y., and Reiter, A. "Algorithms for fast evaluation of Boolean expressions," Acta Inf. **4** (1975), 107-116. Breitbart, Y., and Reiter, A. "A branch-and-bound algorithm to obtain an optimal evaluation tree for monotonic Boolean functions," Acta Inf. **4** (1975), 311-319. Breitbart, Y., and Gal, S. "Analysis of algorithms for the evaluation of monotonic Boolean functions," IEEE Trans. Comp. **TC-27**, 11 (1978), 1083-1087. Brown, P. J. "Functions for selecting tests in diagnostic key construction," Biometrika 64 (1977), 589-596. Cerny, E., Mange, D., and Sanchez, E. "Synthesis of minimal binary decision trees," IEEE Trans. Comp. **TC-28**, 7 (1979), 472-482. Cerny, E., and Moreau, T. "Test evaluation with a decision machine", in Proc. 9th IEEE Fault-Tolerant Comp. Symp., 1979, 117-120. Chang, H. Y. "An algorithm for selecting an optimal set of diagnostic tests," IEEE Trans. Comp. **EC-14**, 5 (1965), 706-711. Chang, H. Y., Manning, E., and Metze, G. Fault Diagnosis of Digital Systems, John Wiley & Sons, New York, 1970. Dallwitz, M. J. "A flexible computer program for generating identification keys," Systematic Zoology 23 (1974), 50-57. Dattatreya, G. R., and Sarma, V. V. S. "Bayesian and decision tree approaches for pattern recognition including feature measurement costs," IEEE Trans. Patt. Anal. & Mach. Intell. **PAMI-3**, 3 (1981), 293-298. Davio, M., Deschamps, J.-P., and Thayse, A. Discrete and Switching Functions, McGraw-Hill, New York, 1978. Egler, J. F. "A procedure for converting logic table conditions into an efficient sequence of test instructions," Commun. ACM 6, 9 (1963), 510-514. Fortune, S., Hopcroft, J., and Schmidt, E. M. "The complexity of equivalence and containment for free single variable schemes," Lecture Notes in

Comp. Sc. **62**, Springer, Berlin, 1978, 227-240. Ganapathy, S., and Rajamaran, V. "Information theory applied to the conversion of decision tables to computer programs," Commun. ACM 16, 9 (1973), 532-539. Garey, M. R. "Optimal binary decision trees for diagnostic identification problems," Ph.D. Thesis, Univ. of Wisconsin, 1970. Garey, M. R. "Optimal binary identification procedures," SIAM J. Appl. Math. **23**, 2 (1972), 173-186. Garey, M. R. "Simple binary identification problems," IEEE Trans. Comp. **TC-21**, 6 (1972), 588-590. Garey, M. R., and Graham, R. L. "Performance bounds on the splitting algorithm for binary testing," Acta Inf. **3** (1974), 347-355. Garey, M. R., and Johnson, D. S. Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, San Francisco, 1979. Gower, J. C., and Barnett, J. A. "Selecting tests in diagnostic keys with unknown responses," Nature 232 (1971), 491-493. Gower, J. C., and Payne, R. W. "A comparison of different criteria for selecting binary tests in diagnostic keys," Biometrika 62 (1975), 665-672. Gyllenberg, H. G. "A general method for deriving determination schemes for random collections of microbial isolates," Ann. Acad. Sc. Fenn. A IV, 69 (1963), 1-23. Halpern, J. "Evaluating Boolean functions with random variables," Int. J. Syst. Sc. **5**, 6 (1974), 545-553. Hanani, M. Z. "An optimal evaluation of Boolean expressions in an online query system," Commun. ACM 20, 5 (1977), 344-347. Harrison, M. A. Introduction to Switching and Automata Theory, McGraw-Hill, New York, 1965. Hartmann, C. R. P., Varshney, P. K., Mehrotra, K. G., and Gerberich, C. L. "Application of information theory to the construction of efficient decision trees," IEEE Trans. Inf. Theory IT-28, 4 (1982), 565-577. Hauska, H., and Swain, P. "The decision tree classifier: design and potential," in Proc. 2nd IEEE Symp. on Machine Proc. of Remotely Sensed Data, Purdue, 1975, 1A38-1A48. Horowitz, E., and Zorat, A. "The binary tree as an interconnection network: applications to multiprocessor systems and to VLSI," IEEE Trans. Comp. **TC-30**, 4 (1981), 247-253. Hulme, B. L., and Worrell, R. B. "A prime implicant algorithm with factoring," IEEE Trans. Comp. **TC-24**, (1975), 1129-1131. Hung, T. Q. "La construction des clefs de diagnostic," M.S. Thesis, University of Montreal, 1974. Hyafil, L., and Rivest, R. L. "Constructing optimal binary decision trees is NP-complete," Inf. Proc. Letters 5, 1 (1976), 15-17. Ianov, I. "The logical schemes of algorithms," in Problems and Cybernetics, Vol. 1, Pergamon Press, New York, 1960, 82-140. Jardine, N., and Sibson, R. Mathematical Taxonomy, John Wiley & Sons, New York, 1971. Johnson, S. M. Optimal sequential testing, Project RAND Res. Mem. RM-1652, 1956. Kanal, L. N. "Problem-solving models and search strategies for pattern recognition," IEEE Trans. Patt. Anal. & Mach. Intell. **PAMI-1**, 2 (1979), 193-201. Kandel, A., and Francioni, J. H. "On the properties and applications of fuzzy-valued switching functions," IEEE Trans. Comp. **TC-29**, 11 (1980), 986-993. King, P. J. H., and Johnson, R. G. "Some comments on the use of ambiguous decision tables and their conversion to computer programs," Commun. ACM 16, 5 (1973), 287-290. Kletsky, E. J. "An application of the information theory approach to failure diagnosis," IRE Trans. Rel. and Qual. Ctrl RQC-9 (1960), 29-39. Knuth, D. E. "Mathematical analysis of algorithms," in Proc. IFIP Congress 71, Vol. I (1971), 135-143. Knuth, D. E. The Art of Computer Programming, Volume 3: Searching and Sorting, Addison-Wesley, Reading, Mass., 1973. Kulkarni, A. V., and Kanal, L. N. "An optimization approach to hierarchi-

cal classifier design," in Proc. 3rd Int'l Joint Conf. on Patt. Rec., San Diego, 1976. Kulkarni, A. V. "On the mean accuracy of hierarchical classifiers," IEEE Trans. Comp. **TC-27**, 8 (1978), 771-776. Lawler, E., and Wood, D. "Branch-and-bound methods: a survey," Oper. Res. **14**, 4 (1966), 699-719. Lee, C. Y. "Representation of switching circuits by binary-decision programs," Bell Syst. Tech. J. **38** (1959), 985-999. Loveland, D. W. Selecting optimal test procedures from incomplete test sets, Duke Univ. Tech. Rep. CS 1979-7. Manber, U., and Tompa, M. "Probabilistic, nondeterministic, and alternating decision trees," Proc. 14th Symp. on Theory of Comp. (1982), 234-244. Mandelbaum, D. "A measure of efficiency of diagnostic tests upon sequential logic," IEEE Trans. Comp. **EC-13** (1964), 630. Martelli, A., and Montanari, U. G. "Optimizing decision trees through heuristically guided search," Commun. ACM 21, 12 (1978), 1025-1039. Masek, W. J. "Some NP-complete set covering problems" (to appear in *Theor. Comp. Sc.*). Meisel, S. W., and Michalopoulos, D. A. "A partitioning algorithm with application in pattern classification and the optimization of decision trees," IEEE Trans. Comp. **TC-22**, 1 (1973), 93-103. Metzner, J. R., and Barnes, B. H. Decision Table Languages and Systems, Academic Press, New York, 1977. Michalski, R. S. Designing extended-entry decision tables and optimal decision trees using decision diagrams, Univ. of Illinois at Urbana-Champaign Tech. Rep. UIUCDCS-R-78-898 (also IEEE Comp. Rep. R78-126), 1978. Misra, J. "A study of strategies for multistage testing," Ph.D. Thesis, The Johns Hopkins Univ., 1972. Moller, F. "Quantitative methods in the systematics of the Actinomycetales IV. The theory and application of a probabilistic identification key," Giorn. Microbiol. **10** (1962), 29-47. Montalbano, M. "Tables, flow charts, and program logic," IBM Syst. J. **1** (1962), 51-63. Moret, B. M. E. "The representation of discrete functions by decision trees: aspects of complexity and problems of testing," Ph.D. thesis, Univ. of Tennessee, Knoxville, 1980. Moret, B. M. E., Thomason, M. G., and Gonzalez, R. C. "The activity of a variable and its relation to decision trees," ACM Trans. Progr. Lang. & Syst. TOPLAS 2, 4 (1980), 580-595. Moret, B. M. E., Thomason, M. G., and Gonzalez, R. C. "The use of activity in testing digital and analog systems," in Proc. 1st IEEE ATPG Workshop, Philadelphia, 1981, 120-127. Moret, B. M. E., Thomason, M. G., and Gonzalez, R. C. Optimization criteria for decision trees, Univ. of New Mexico Tech. Rep. CS81-6. Moret, B. M. E., and Shapiro, H. D. Experience with the minimum test set problem, Univ. of New Mexico Tech. Rep. CS82-4. Morse, L. E. "Specimen identification and key construction with time-sharing computers," Taxon 20 (1971), 269-282. Osborne, D. V. "Some aspects of the theory of dichotomous keys," New Phytol. **62** (1963), 111-160. Pankhurst, R. J. "A computer program for generating diagnostic keys," Comp. J. **13**, 2 (1970), 145-151. Payne, H. J., and Meisel, W. S. "An algorithm for constructing optimal binary decision trees," IEEE Trans. Comp. **TC-26**, 9 (1977), 905-916. Payne, R. W. "Reticulation and other methods of reducing the size of printed diagnostic keys," J. Gen. Microbiol. **98** (1977), 595-597. Payne, R. W., and Preece, D. A. "Identification keys and diagnostic tables: a review," J. Royal Statist. Soc. A 143, 3 (1980), 253-292 (with discussion). Payne, R. W. "Selection criteria for the construction of efficient diagnostic keys," J. Statist. Planning & Inf. **5** (1981), 27-36. Perl, Y., and Breitbart, Y. "Optimal sequential arrangement of evaluation trees for Boolean functions," Inf. Sc. **11** (1976), 1-12. Picard, C.

F. Graphes et Questionnaires. Tome 2: Questionnaires, Gauthier-Villars, Paris, 1972. Pollack, S. L. "Conversion of limited-entry decision tables to computer programs," Commun. ACM 8, 11 (1965), 677-682. Pooch, U. W. "Translation of decision tables," Comp. Surveys 6 (1974), 125-151. Prather, R. E., and Casstevens, H. T. "Realization of Boolean expressions by atomic digraphs," IEEE Trans. Comp. **TC-27**, 8 (1978), 681-688. Press, L. I. "Conversion of decision tables to computer programs," Commun. ACM 8, 6 (1965), 385-390. Rabin, J. "Conversion of limited-entry decision tables into optimal decision trees: fundamental concepts," SIGPLAN Notices 6, 8 (1971), 68-74. Reingold, E. M. "On the optimality of some set algorithms," J. ACM 19, 4 (1972), 649-659. Reinwald, L. T., and Soland, R. M. "Conversion of limited-entry decision tables to computer programs I: minimum average processing time," J. ACM 13, 3 (1966), 339-358. Reinwald, L. T., and Soland, R. M. "Conversion of limited-entry decision tables to computer programs II: minimum storage requirements," J. ACM 14, 4 (1967), 742-755. Rescigno, A., and Maccacaro, G. A. "The information content of biological classifications," in Information Theory: Fourth London Symposium, Cherry, C., Ed., Butterworths, London, 1961, 437-445. Riesel, H. "In which order are different conditions to be examined," BIT 3 (1963), 255-256. Rivest, R. L., and Vuillemin, J. On the number of argument evaluations required to compute Boolean functions, U.C. Berkeley Elec. Res. Lab. Mem. ERL-M476, 1976. Rivest, R. L., and Vuillemin, J. "On recognizing graph properties from adjacency matrices," Theor. Comp. Sc. **3** (1976), 371-384. Rounds, E. M. "A combined non-parametric approach to feature selection and binary tree design," in Proc. IEEE Conf. Patt. Rec. & Image Proc., Chicago, 1979, 38-43. Savage, J. E. The Complexity of Computing, John Wiley & Sons, New York, 1976. Schumacher, H., and Sevcik, K. C. "The synthetic approach to decision table conversion," Commun. ACM 19, 6 (1976), 343-351. Sethi, I. K., and Chatterjee, B. "Conversion of decision tables to efficient sequential testing procedures," Commun. ACM 23, 5 (1980), 279-285. Shannon, C. E. "A mathematical theory of communication," Bell Syst. Tech. J. **27** (1948), 379-423. Shapiro, H. D. Private communication. Univ. of New Mexico, 1981. Shaw, C. J., Ed. "Decision tables," SIGPLAN Notices 6, 8 (1971). Shwayder, K. "Extending the information theory approach to converting limited-entry decision tables to computer programs," Commun. ACM 17, 9 (1974), 532-537. Shwayder, K. "Combining decision rules in a decision table," Commun. ACM 18, 8 (1975), 476-480. Slagle, J. R. "An efficient algorithm for finding certain minimum cost procedures for making binary decisions," J. ACM 11, 3 (1964), 253-264. Slagle, J. R., Chang, C. L., and Lee, R. C. T. "A new algorithm for generating prime implicants," IEEE Trans. Comp. **TC-19**, 4 (1970), 304-310. Slagle, J. R., and Lee, R. C. T. "Application of game tree searching techniques to sequential pattern recognition," Commun. ACM 14, 2 (1971), 103-110. Tabloski, T. F., and Maule, F. J. "Numerical expansion technique for minimal multiplexer logic circuits," IEEE Trans. Comp. **TC-25**, 7 (1976), 684-702. Thayse, A., Davio, M., and Deschamps, J. P. "Optimization of multivalued decision algorithms," in Proc. Int. Symp. on Multival. Logic, Rosemont, 1978, 171-178. Thayse, A. "P-functions: a new tool for the analysis and synthesis of binary programs," IEEE Trans. Comp. **TC-30**, 2, (1981), 126-134. Thayse, A. Boolean Calculus of differences, Lecture Notes in Comp. Sc. 101, Springer Verlag, Berlin, 1981. Voith,

R. "ULM implicants for minimization of universal logic modules circuits," IEEE Trans. Comp. **TC-26**, Voss, E. G. "The history of keys and phylogenetic trees in systematic biology," J. Sci. Denison Univ. **43** (1952), 1-25. Weide, B. "A survey of analysis techniques for discrete algorithms," Comp. Surveys 9, 4 (1977), 291-313. Willcox, W. R., Lapage, S. P., and Holmes, B. "A review of numerical methods in bacterial identification," Antonie van Leeuwenhoek 46, 3 (1980), 233-299. Wong, E., and Youssefi, K. "Decomposition--a strategy for query processing," ACM Trans. Database Syst. **TODS 1**, 3 (1976), 223-241. Worrell, R. B. "Using the Set Equation Transformation System in fault tree analysis," in Reliability and Fault Tree Analysis, Barlow, R. E., et al., Eds., SIAM Press, Philadelphia, PA, 1975, 165-186. Wu, C., Landgrebe, D., and Swain, P. The decision tree approach to classification, Purdue Univ. Tech. Rep. TR-EE 75-17, Lafayette, IN, 1975. Yasui, T. "Some aspects of decision table conversion techniques," SIGPLAN Notices 6, 8 (1971), 104-111. You, K. C., and Fu, K. S. "An approach to the design of a linear binary tree classifier," Proc. 3rd IEEE Symp. Mach. Proc. of Remotely Sensed Data, Purdue, 1976, 3A1-3A10. Zimmerman, S. "An optimal search procedure," Am. Math. Monthly 66 (1959), 690-693. iv. Tech. Rep. TR-EE 75-17, Lafayette, IN, 1975. Yasui, T. "Some aspects of decision table conversion techniques," SIGPLAN Notices 6, 8 (1971), 104-111. You, K. C., and Fu, K. S. "An approach to the design of