COBWEB/3: A Portable Implementation

KATHLEEN MCKUSICK
KEVIN THOMPSON

Sterling Software/AI Research Branch
NASA Ames Research Center, Mail Stop 244-17
Moffett Field, California 94035 USA

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-90-6-18-2

June 20, 1990

COBWEB/3: A Portable Implementation

Kathleen McKusick
Kevin Thompson

Sterling Software/AI Research Branch
NASA Ames Research Center, Mail Stop 244–17
Moffett Field, CA 94035 USA

E-mail: LABYRINTH@PTOLEMY.ARC.NASA.GOV

Version 1.1
September 17, 1990

Abstract

This manual documents Cobweb/3, a portable implementation of an algorithm for data clustering and incremental concept formation. The manual gives an overview of the Cobweb/3 system and the algorithm on which it is based, as well as the practical details of obtaining and running the system code. The implementation features a flexible user interface which includes the ability to display graphically the concept hierarchies that the system constructs.

The ideas in this paper have resulted from work with the other members of the ICARUS project: Pat Langley, John Allen, John Gennari, Wayne Iba, and Deepak Kulkarni. Many of them have contributed code to this implementation, as did Patrick Young. Doug Fisher contributed important ideas.

Contents

1. Introduction

COBWEB/3 is a tool for organizing descriptions of objects, places, events, or any data that can be represented as conjunctions of attributes and their values. Because it forms classes from its input, the system can be useful in clustering data into groups of instances that share regularities across a number of attributes.

COBWEB was originally developed as a model of incremental concept formation, to demonstrate some psychological aspects of this process in humans. After observing the world, humans form concepts by organizing their observations on the basis of shared characteristics among things they observe. For example, the general concept of dog is built up incrementally over time after many experiences with particular dogs of varying appearance and behavior. The concept grows in generality and becomes more useful as one sees more dogs. It becomes possible to discriminate dogs from other animals that look very similar. Eventually one can make accurate predictions about a dog one is encountering for the first time: it probably barks, could bite if angered, has a cold wet nose. Humans can form concepts despite irrelevant information (dog X is standing on a red carpet) and incomplete examples (dog Y's owner is obstructing its tail from view). Much human concept learning occurs through this process of organizing observations into general classes without the advice of a tutor.

Similarly, COBWEB takes observations presented to it and organizes them, forming concepts that summarize the instances they cover. As in human concept formation, this process occurs incrementally: concepts become more discriminating over time as the system encounters more examples. Concepts form despite irrelevant information and incompleteness in the incoming observations, and they form without the advice of a tutor. Like humans, COBWEB uses concepts to classify observations and make predictions about what it observes when observed information is incomplete.

COBWEB's approach to classification and learning is known as *conceptual clustering* (Michalski & Stepp, 1983; Fisher & Langley, 1986). Five key traits, which COBWEB shares with a few other systems (Feigenbaum, 1963; Kolodner, 1984; Lebowitz, 1987), distinguish it from most other work in machine learning.

1. *Hierarchical organization of concepts.* Instead of forming flat classes that merely group similar instances together as do iterative optimization algorithms (Anderberg, 1973; Cheeseman et al., 1988), COBWEB forms a *concept hierarchy.* In the hierarchy, nodes represent concepts that are partially ordered by generality. Each node includes a concept description, a list of the probabilities that certain features will occur in instances covered by the concept. We describe the hierarchical organization in more detail in Section 2.1.
2. *Top-down classification.* COBWEB classifies a new instance by sorting it down through the concept hierarchy to locate the classes in which the instance belongs. Typically an instance is incorporated into a series of concepts, first into a very general concept, then into increasingly specific ones.
3. *Unsupervised Learning.* COBWEB uses no tutor to guide classification or provide feedback after classification has occurred, so it does not require predefined class information in its input.

Instead it forms its own classes on the basis of shared characteristics (and dissimilarities) in the observations it receives.

4. *Incremental Learning.* Some systems (Cheeseman et al., 1988; Michalski & Stepp, 1983) require all input before they can partition observations into classes, or accept instances one at a time but then reprocess previously seen instances. COBWEB processes one instance at a time and never does massive reprocessing of instances.
5. *Hill climbing.* COBWEB makes adjustments to an existing concept hierarchy based on new knowledge—the input it is currently processing—rather than keeping several alternative hierarchies in memory. This can be viewed as a form of hill-climbing search: each new hierarchy is a function of the previous hierarchy, an incoming instance, and some operator. Since no alternative hierarchies are considered or saved, memory requirements are limited to what is needed to store a single hierarchy. This is in contrast to memory-intensive search techniques such as depth-first or breadth-first search, which would require storage of multiple hierarchies.

Beyond clustering instance descriptions, COBWEB forms general descriptions of classes at various levels of abstraction. The system makes available to the user both these general descriptions and the particular instances that comprise a class. Since COBWEB can use the classes it creates to predict attributes missing from new input, the system can be used in prediction tasks like those for which decision tree classifiers are useful, such as medical diagnosis or fault identification. However, unlike algorithms for inducing decision trees (Quinlan, 1986), COBWEB does not assume that instances are preclassified. As mentioned above, it determines its own classes without requiring a tutor or any explicit class information.

In addition to its role as a classifier, the system can also be viewed as an efficient case-based memory system. By creating a hierarchy of abstract classes, COBWEB indexes and allows the efficient retrieval of the individual instances it has seen as well as abstract concepts it uses to summarize the instances. The system uses the abstractions it has derived not only to index stored instance descriptions but also to make predictions about incoming cases that may differ from any particular case it has seen before. Therefore COBWEB moves beyond the observed cases by retrieving and making use of relevant abstract case descriptions.

This manual gives an overview of the COBWEB/3 system and the algorithm on which it is based (Section 2), as well as the practical details of obtaining and running the system code (Section 3). The algorithm we describe is a hybrid of Fisher's (1987a) COBWEB algorithm and Gennari's (1989) CLASSIT algorithm. In Section 2.6 we explain how this implementation is tied to each. Since the manual is intended as a practical guide to using the system, our treatment of theoretical issues is brief. For the reader interested in more detail, we recommend Fisher (1987a, 1987b), Gennari et al. (1989), Gennari (1989a, 1990), and Fisher and Langley (in press), which discuss the theory behind COBWEB in depth. For information about how COBWEB has been incorporated into other research projects and systems, we recommend Langley et al. (1989), Iba and Gennari (in press), Thompson and Langley (1989), and Yoo, Yang, and Fisher (in press). COBWEB's evaluation function, category utility, is discussed at length in Gluck and Corter (1985).

(a)

DIAMETER	medium
COLOR	green
REBOUND	high

(b)

DIAMETER	2.52
COLOR	3
REBOUND	0.71

Figure 1. Two instances of a tennis ball: (a) nominal attributes, (b) numeric attributes.

2. An Approach to Concept Formation

Now we turn to the details of how concept formation works in COBWEB. Any concept formation system needs some way both to represent knowledge, and to organize and store this knowledge in memory. We discuss this in Section 2.1. Since COBWEB is an incremental hill-climbing system, it relies on a set of operators to create new knowledge hierarchies from old. We discuss these operators in Section 2.2. To decide which operator to apply at any given time, COBWEB evaluates the alternatives using an evaluation function which we describe in Section 2.3. How these pieces fit together in COBWEB's algorithm for classification, learning, and prediction is the subject of Section 2.4. Finally, we illustrate the algorithm with a sample execution in Section 2.5.

2.1 Representation and Organization of Knowledge

COBWEB accepts as input a series of instance descriptions. Instances are represented as an ordered set of attribute-value pairs, and each attribute may take on only one value. The implementation of COBWEB described here can handle either nominal or numeric attribute values.

To illustrate how instances are represented, Figure 2.1 shows a tennis ball described in terms of nominal attributes (a) and numeric attributes (b). The attributes chosen to describe this instance are SIZE, COLOR, and REBOUND. In the nominal case, SIZE of a ball can be described as *small*, *medium*, or *large*, COLOR as *green*, *red*, etc., REBOUND as *low*, *medium*, or *high*. In the numeric case, SIZE can be described as the diameter of the ball, COLOR as a value on a continuous numeric scale from 0 to 10, and REBOUND as a percentage of the dropped height when released onto a smooth surface.

As we have noted, COBWEB uses instances like those illustrated to construct and maintain a concept hierarchy. COBWEB adds each instance it receives to the hierarchy, adding knowledge by changing information within the concept nodes and in some cases changing the overall structure of the hierarchy. The concept hierarchy is a tree, with each node in the tree describing a concept. The concepts are partially ordered, from most general (the root of the tree, summarizing all objects) to most specific (the leaves of the tree, specific instances).

Each concept node describes a class of instances. Like instances, concept nodes are described in terms of attributes and values, as this is a natural way to summarize the instances covered by the concept. The information stored at a concept node is slightly different when attribute values are nominal in the instance descriptions than it is when numeric values are used. In both cases,

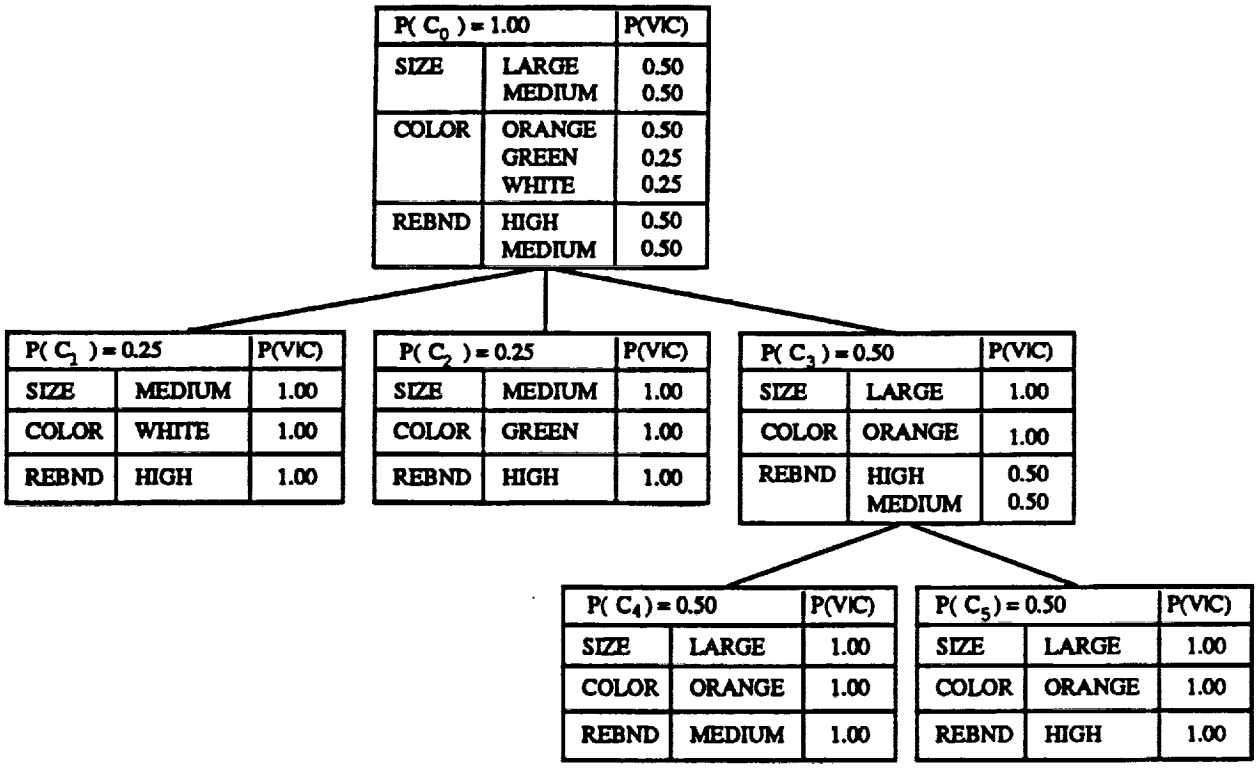

Figure 2. COBWEB hierarchy of ball instances with nominal attribute values.

COBWEB stores the probability of the concept's occurrence at each concept node (Smith & Medin, 1983). It also stores information about every attribute observed in the instances that are covered by the concept. In the nominal case, COBWEB stores the conditional probability of each attribute value, given membership in the class covered by the concept. In the numeric case, the system stores a continuous normal distribution for each attribute, in the form of a mean and a standard deviation.

Figure 2.1 shows a sample COBWEB hierarchy of instances with nominal attribute values. This hierarchy describes instances of balls that differ in their size, color, and amount of rebound. The nodes in the hierarchy are numbered (C_0, ..., C_5) in the order that COBWEB generated them. At the top of each node is its probability of occurrence, specified with respect to its parent. For example, if an instance belongs in the class C_3, then the probability that it belongs in the class C_4 is 0.5. Below the class probability is the list of attributes and their possible values. As in the instance descriptions, the attributes which describe the instances stored in this hierarchy are SIZE, COLOR, and REBOUND.

Consider the SIZE attribute. Two values have been seen so far, *medium* and *large*. Balls that are members of the more general root node, which describes four instances, have an equal chance of being *medium* or *large*, as indicated in the $P(V|C)$ column. Balls that are members of the more specific node C_3, which describes two instances, are all *large*. The terminal nodes in the hierarchy,

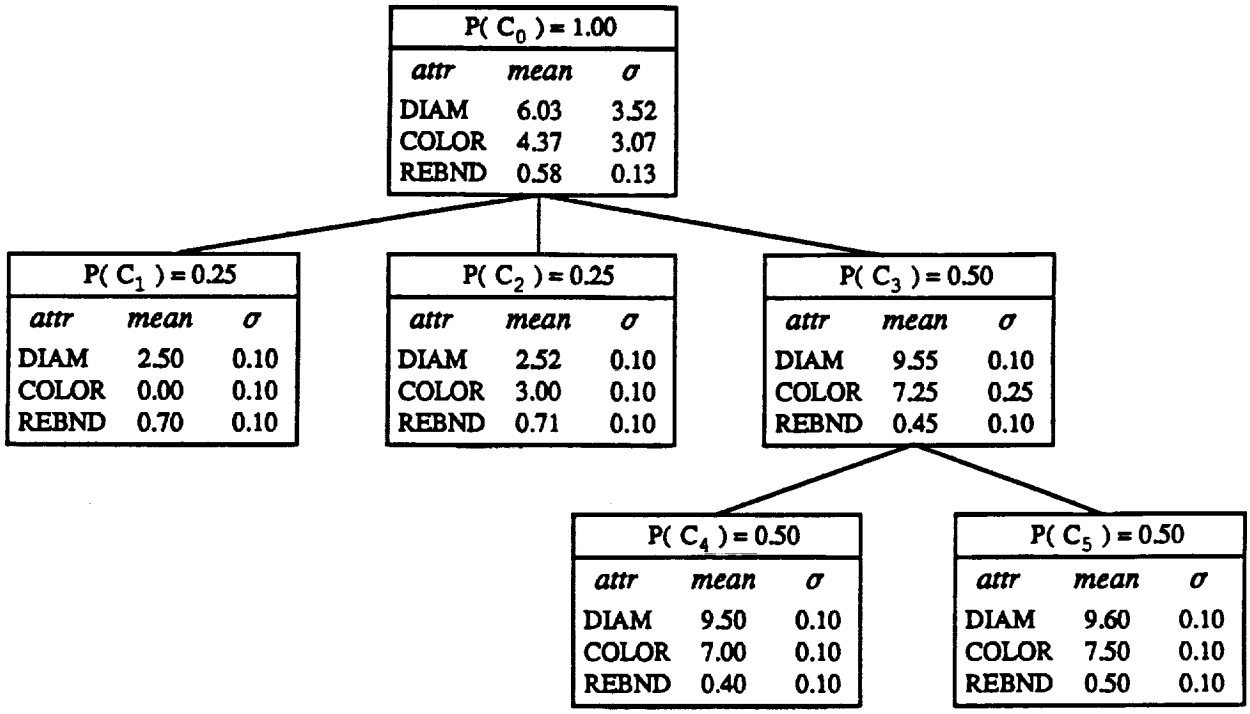

Figure 3. COBWEB hierarchy of ball instances with numeric attribute values.

C_1, C_2, C_4, and C_5, all describe a single instance. The probabilistic descriptions are simple in these nodes: each attribute has one value that is certain to occur.

Figure 2.1 shows the hierarchy that COBWEB forms from similar instances described with numeric attribute values. The node headers show the conditional probabilities of the concept's occurrence, just as in those in the previous hierarchy. The other information stored at the node is somewhat different. Instead of a list of possible values for each attribute, each node contains the mean and standard deviation of the values seen in the instances covered by the node. Note that except at the root node, the standard deviations are all 0.1, a minimum value. We will return to an explanation of this minimum value in Section 2.3 when we discuss the acuity parameter.

2.2 Operators for Classification and Learning

COBWEB relies on four operators to incorporate the knowledge in an instance description into the concept hierarchy. At each level of the concept hierarchy, the system applies an operator that uses a particular mechanism to classify an object into the hierarchy at that level. The operators apply locally to the subtree composed of the last concept node to which an instance was classified (we will refer to this as the "current node"), and the children of this node. Initially, all instances are classified to the root node, making the root the current node. COBWEB selects which operator to apply by evaluating alternate classifications with an evaluation function, discussed in Section 2.3.

COBWEB's operators are illustrated in Figure 2.2. COBWEB applies the first of the four operators,

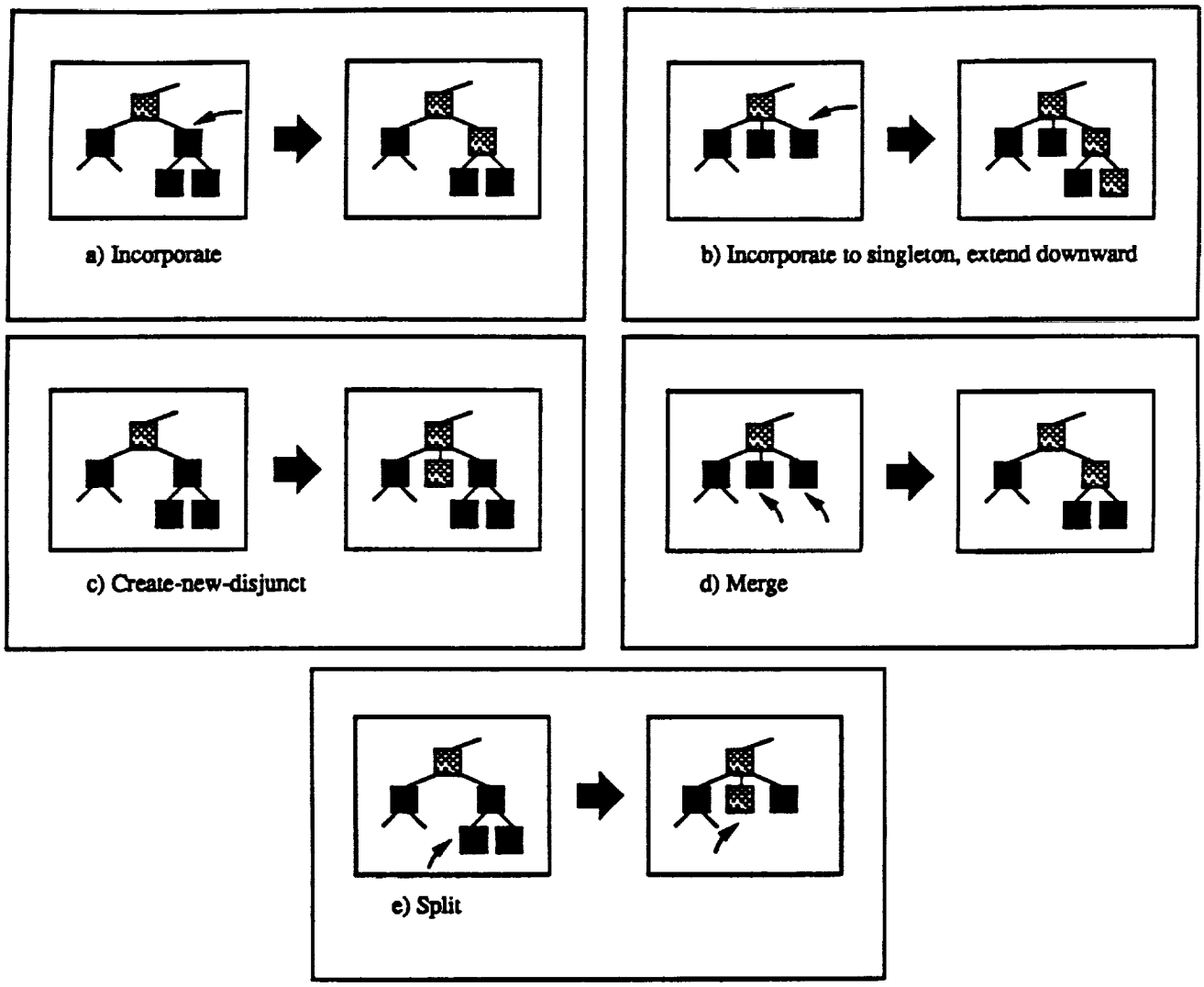

Figure 4. COBWEB's learning operators.

Incorporate, when an instance fits into an existing concept well. This operator integrates the instance into one of the child nodes. If this child node is not a singleton (i.e. it describes more than one instance), COBWEB updates the conditional probabilities for the concept and each of the attribute values. Figure 2.2 (a) illustrates this process schematically. (For a more precise definition of Incorporate and the other operators, refer to the pseudocode in Appendix A.) If the node to which COBWEB is incorporating the instance is a singleton, COBWEB must extend the hierarchy downward, as shown in Figure 2.2(b).

COBWEB applies the second operator, *Create-new-disjunct*, when an instance has very different characteristics from any existing concept at the current level, as determined by its evaluation function. This operator places the instance in a category by itself, a sibling of the existing concept nodes. Figure 2.2 (c) shows this process.

Two additional operators allow COBWEB to restructure the hierarchy without reprocessing previous instances. These give the system some power to reorganize in light of new incoming knowledge while remaining an incremental system. COBWEB applies the *Merge* operator when the hierarchy is overly branched, and combining two classes provides a good concept to which to classify the incoming instance. The Merge operator merges two child nodes and incorporates the instance into this new combined class. The effects of this operator are shown in Figure 2.2 (d).

COBWEB applies the *Split* operator when the hierarchy contains a node that is too general and therefore less useful for classification and prediction. In this case breaking the general node into well-defined classes creates a good match for the incoming instance and distinguishes classes that turn out to be too different to be grouped at this level. The Split operator breaks the current node up into several distinct concept nodes, as illustrated in Figure 2.2 (e). This split is achieved by removing the current node and replacing it with its children. Then COBWEB incorporates the instance into one of these more specific nodes.

2.3 Evaluation function

Note that we have not yet explained how COBWEB selects which operators to apply as it sorts an instance down through each level of the hierarchy. To make this choice, the system must be able to evaluate alternative classifications and apply the operator which produces the best. COBWEB uses an evaluation function, *category utility* (Gluck & Corter, 1985), to score these alternatives.

Different classifications of a new instance result in a number of different *partitions* of all the instances into classes. Category utility gives a high score to partitions which maximize similarity among class members (intra-class similarities) and differences between members of different classes (inter-class differences). Intuitively, it makes sense to form classes whose members have very similar attributes, which in turn differ greatly from the attributes of members of other classes.

In effect, category utility trades off the *predictiveness* of each attribute value (the probability of an instance's membership in a class, given its attribute value) and the *predictability* of the value (the probability of the value, given that an instance is a member of a class). For example, if you know an instance can be classified as a bird, you can predict that it has wings. Yet wings may not be a terribly predictive attribute value, as bats, insects, and airplanes are all classes that also contain instances with wings. Even if you know that an instance has wings, you cannot predict with confidence that it is a bird from this attribute.

Conversely, if you know that an instance has an understanding of quantum mechanics, you can predict that it is human and not a member of another species. Therefore this attribute value is predictive. Yet this is not a very predictable attribute value, as knowing that an instance is human does not necessarily mean that it understands quantum mechanics.

To summarize, predictive values are those most nearly unique to a certain class and therefore indicative of it. The evaluation function favors classes with many predictive attribute values because these maximize inter-class differences. Predictable values are those that many members share and therefore are easy to guess accurately. The evaluation function favors classes with many predictable values because these maximize intra-class similarities. Since attributes are often not

both predictable and predictive, category utility trades off the two, maximizing each as much as possible. The category utility equation can be summarized as

$$\frac{X - Y}{K},$$

where X is the expected number of attribute values that can be correctly guessed, given the K categories, and Y is the expected number of attribute values that can be correctly guessed without any category knowledge. Dividing by K, the total number of classes, normalizes for partitions with differing numbers of classes. In the expanded equation, the X term is

$$\sum_{k=1}^{K} P(C_k) \sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij} | C_k)^2,$$

summing across K classes, I attributes, and J values. $P(C_k)$ is the probability of occurrence of a particular class C_k and $P(A_i = V_{ij}|C_k)$ is the conditional probability of a particular value V_{ij} given membership in the class. The Y term expands to

$$\sum_{i=1}^{I} \sum_{j=1}^{J} P(A_i = V_{ij})^2,$$

where $P(A_i = V_{ij})$ is the probability of a particular value at the parent of the node classes being considered; that is, the probability across all classes without category knowledge. The complete equation is:

$$\frac{\sum_{k=1}^{K} P(C_k) \sum_i \sum_j P(A_i = V_{ij}|C_k)^2 - \sum_i \sum_j P(A_i = V_{ij})^2}{K}.$$

For information on the derivation of this equation, refer to Gluck and Corter (1985), which gives a two-class version of the equation, and Fisher (1987b), which gives this multi-class form.

COBWEB applies this version of category utility when instances have nominal attributes. As is, it cannot be applied when instances have numeric attributes, since it is unable to distinguish any difference between numbers that are close in value from those that are far apart. For example, the real numbers 3.112, 3.113, and 12.9 would all be treated as distinct, unrelated values by the original equation. However, category utility can be adapted to deal with numeric valued attributes. Since probabilities for numeric attributes are stored as a normal distribution (a mean and a standard deviation), the innermost summation in the ordinary category utility equation can be replaced with the integral of the equation for the normal distribution

$$\sum_{j}^{values} P(A_i = V_{ij})^2 \quad \Leftrightarrow \quad \int \frac{1}{\sigma^2 2\pi} e^{-(\frac{x-\mu}{\sigma})^2} \, dx \quad = \quad \frac{1}{\sigma} \frac{1}{4\sqrt{\pi}}.$$

The transformed evaluation function is then

$$\frac{\sum_{k}^{K} P(C_k) \sum_{i}^{I} 1/\sigma_{ik} \; - \; \sum_{i}^{I} 1/\sigma_{ip}}{4K\sqrt{\pi}} \; ,$$

where K is the number of classes, I is the number of attributes, σ_{ik} is the standard deviation for attribute i in class k, σ_{ip} is the standard deviation for attribute i in the parent (i.e. where no class information is present).

One problem with this transformed equation is that $\sigma = 0$ when a concept node describes a single instance, so the $1/\sigma$ is ∞ in this case. In this situation, COBWEB relies on a user-specified parameter, *acuity*, to serve as a minimum value for σ. Acuity represent the minimum detectable difference between instances.

Typically, some instance descriptions are incomplete, with values missing for one or more attributes. In this implementation of COBWEB, we adapt the category utility equations so they handle this situation by dividing the attribute summations by I, the number of attributes in the incoming instance. The revised equations are:

$$\frac{\sum_{k=1}^{K} P(C_k) \frac{\sum_i^I \sum_j^J P(A_i=V_{ij}|C_k)^2}{I} - \frac{\sum_i^I \sum_j^J P(A_i=V_{ij})^2}{I}}{K}$$

for discrete values, and

$$\frac{\sum_{k=1}^{K} P(C_k) \frac{\sum_i^I 1/\sigma_{ik}}{I} \; - \; \frac{\sum_i^I 1/\sigma_{ip}}{I}}{4K\sqrt{\pi}}$$

for continuous values. As Gennari (1989) points out, mixing nominal and numeric attributes in a single instance description is an open issue in the literature on numerical taxonomy and clustering. However, Gennari (1990) presents evidence that summing together terms from both forms of the equation works well in domains with mixed data. We include the capability to handle instances with mixed attribute types in COBWEB/3.

2.4 The COBWEB Algorithm

We now turn to a discussion of how the evaluation function, operators, and concept hierarchy work together in the COBWEB algorithm. Theoretically, COBWEB has two modes of operation, *learning* mode and *prediction* mode. In our implementation, the two modes are not completely distinct, but it is helpful to define learning and prediction in terms of their impact on the COBWEB hierarchy before we discuss the particulars of the implementation. In learning mode, COBWEB classifies each instance and incorporates it permanently into the hierarchy, changing the hierarchy's structure and thus affecting future classification and prediction. In prediction mode, COBWEB also classifies an instance but without incorporating it into the hierarchy, merely locating the most specific concept

that describes it. Then COBWEB uses the probabilistic information stored at this concept node to predict attributes missing from the instance description. Strictly speaking, learning and prediction are different processes, although both rely on the same classification mechanism. Learning alters the hierarchy while prediction does not.

In this implementation of COBWEB, prediction and learning are partially intertwined. Learning can run without prediction: COBWEB/3 reads in instances and incorporates them into the hierarchy. However, the prediction mode is automated for running large-scale empirical studies with the system. An instance is read in, then each attribute in turn is excised and COBWEB/3 attempts to predict it, and reports how it does. When the system completes prediction on an object, it subsequently turns learning on and adds the instance to the hierarchy. Thus prediction is distinct from learning but runs in tandem with it. Below we describe the COBWEB/3 algorithm for each processing mode.

2.4.1 LEARNING

COBWEB accepts instance descriptions one at a time. The system actually has two inputs each time it processes an instance, the instance description and an existing concept hierarchy. Typically, the concept hierarchy is one COBWEB has built from previous instances. Alternatively, the concept hierarchy can be one the user has built and passed to the system. The concept hierarchy can also be empty; if COBWEB has not seen any instances yet and no user-defined hierarchy is passed in, the system bootstraps from the first instance, making this instance the root of a new hierarchy.

Given an instance and a hierarchy, COBWEB's task is to create a new hierarchy that incorporates the new, incoming knowledge embodied in the instance description. Simply stated, the learning task is

- *Given*: an instance and a concept hierarchy
- *Do*: create a new hierarchy that incorporates the instance

The algorithm that carries out this learning task can be viewed as a three-step recursive process. There is an initialization step at the root of the hierarchy, and then classification takes place level by level. The steps followed in learning mode are:

start. Initialize at root

step 1. Preview the next level
step 2. Incorporate the instance
step 3. Recurse to the next level

Notice the basic recursive mechanism COBWEB uses to sort an instance down through the hierarchy. Starting at the most general root node, the system previews and incorporates, previews and incorporates, as it sorts instances downward. In so doing, COBWEB selects a path through the tree composed of increasingly specific concept nodes. Now we turn to the details of each step, including the initialization.

(a)

DIAMETER	SMALL
COLOR	RED
REBOUND	LOW

(b)

$P(C_0) = 1.00$		P(VIC)
SIZE	LARGE	0.50
	MEDIUM	0.50
COLOR	ORANGE	0.50
	GREEN	0.25
	WHITE	0.25
REBND	HIGH	0.50
	MEDIUM	0.50

(c)

$P(C_0) = 1.00$		P(VIC)
SIZE	LARGE	0.40
	MEDIUM	0.40
	SMALL	0.20
COLOR	ORANGE	0.40
	GREEN	0.20
	WHITE	0.20
	RED	0.20
REBND	HIGH	0.40
	MEDIUM	0.40
	LOW	0.20

Figure 5. Instance of a marble (a) and a root node before (b) and after (c) incorporation of the marble.

start. Initialize at root

COBWEB begins by incorporating the current instance into the root concept node. As the most general node in the hierarchy, the root summarizes all instances the system has seen so far. Returning to the example in Figure 2.1, the root node is the most general concept for BALL. This concept does not yet capture the variety of most of the balls in the world, only an abstraction of the balls it has seen: two tennis balls and two basketballs. Suppose the system is classifying a new ball description into the hierarchy, that of a marble. First the system must incorporate it into the root, which generalizes the root concept to cover the marble instance. Figure 5 shows the root node before and after the incorporation.

The probability for the root category stays the same (unity), since all instances sort to the root. However, note that the values *small*, *red*, and *large* appear because they now have nonzero probabilities. Furthermore, the probabilities of the other values have shifted slightly. At the SIZE attribute for example, even though there has been no change in the absolute number of *large* instances (two), the probability of this value in the concept shifts from $\frac{2}{4}$ to $\frac{2}{5}$ because the concept covers five instances instead of four. The probability of the *medium* value similarly shifts to $\frac{2}{5}$. The probability of the *small* value is $\frac{1}{5}$. The conditional probability of each value for COLOR and REBOUND also shift.

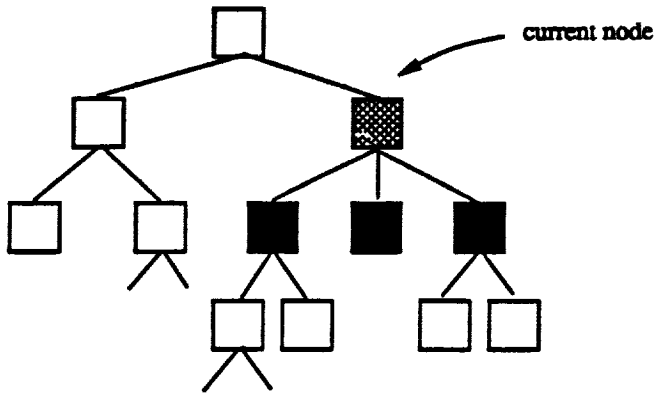

Figure 6. Previewing involves only the current node and its children.

step 1. Preview the next level

During the preview step, COBWEB does a one-step lookahead to determine the best place to incorporate the instance at the current level of the hierarchy. This lookahead does not permanently change the tree; the system simply considers the effect of various classification strategies. Previewing involves only the current node[1] and the immediate children of the current node in the classification decision, as illustrated in Figure 2.4.1. Thus classification decisions are localized to a small area of the hierarchy at a time. While previewing, COBWEB considers two obvious classification strategies that have minimal impact on the structure of the hierarchy:

1. Classification to an existing concept.

   COBWEB tries incorporating the new instance into each of the existing classes, seeking an existing concept that fits the instance well. It evaluates each resulting partition with the evaluation function described in Section 2.3. If one of these partitions gets the best category utility score, application of the Incorporate operator is indicated.

2. Classification as a new concept.

   COBWEB tries setting the instance apart from existing classes, creating a new, disjunct class with the new instance as the only member. If this partition gets the best category utility score, application of the Create-new-disjunct operator is indicated.

COBWEB considers two additional classification strategies that partially reorganize the tree. Since the system processes instances one at a time, it can be misled by the particular order of early instances it receives and build a non-optimal hierarchy. The following methods of incorporation help COBWEB to recover and adapt the hierarchy it constructed with more limited information, to effectively cover both old and new instances.

1. *Current node* refers to the last node where an instance was incorporated. Initially this is the root.

3. Merging two concepts.

   COBWEB tries merging the two best classes, and incorporates the instance into this new combined class. If this partition gets the best category utility score, application of the Merge operator is indicated.

4. Splitting a concept.

   COBWEB tries splitting the best class, replacing this node with the node's children. Then the system tries incorporating into each child of the split node, evaluating each resulting partition with the evaluation function. If this partition gets the best category utility score, application of the Split operator is indicated.

Step 2. Incorporate the instance

Based upon the results of the preview, COBWEB applies one of its four operators to incorporate the instance into the concept hierarchy. Recall from Section 2.2 that we denote COBWEB's four operators *Incorporate*, *Create-new-disjunct*, *Merge*, and *Split*. If incorporating an existing concept has the best category utility, COBWEB applies the Incorporate operator, handing it the instance and a pointer to the node. If making a new disjunct has the best category utility, the system applies the *Create-new-disjunct* operator, handing it the instance and a pointer to the current node (the node from which the disjunct is to be made). If merging two nodes has the best category utility, COBWEB applies the Merge operator, handing it the instance and a pointer to the two nodes to merge. If splitting a node has the best category utility, the system applies the Split operator, handing it the instance, a pointer to the node to split, and a pointer to the best child concept of that node. After applying an operator, which permanently incorporates the instance at the current level, COBWEB continues to the next step, recursing to the levels below.

Step 3. Recurse.

COBWEB repeats from step 1 for the next level down. As the system recurses, it continues to incorporate the instance into concepts of increasing specificity. COBWEB halts when it classifies a new instance into a singleton node at the bottom of the hierarchy, or when it creates a new disjunct with the instance at any level.

2.4.2 PREDICTION

COBWEB can use a concept hierarchy to make predictions about instances it has not encountered before. In particular, it can predict attributes that are missing in partially-described instances. Given a partial description and a hierarchy (one either the system or the user has constructed), COBWEB's task is to classify the instance and predict missing attributes.

In this implementation, prediction is automated for comparative studies between COBWEB/3 and other systems. Partial instances are manufactured by removing one attribute at a time.[2] The system then classifies the instance and predicts the value of the missing attribute. During the next

2. COBWEB and CLASSIT can in theory predict any number of attributes, but COBWEB/3 does not yet support this feature.

iteration, a different attribute is removed. This process continues until COBWEB has predicted each attribute in turn. The system reports the results of this process. To summarize, the prediction task is:

- *Given*: an instance and a concept hierarchy
- *Do*: remove and predict each attribute in the instance

The prediction algorithm is divided into four steps. Iterate for every attribute in the instance:

start. Remove an attribute from the instance

step 1. Preview the next level
step 2. Recurse to the next level

finish. Predict missing attribute

The prediction algorithm, like the learning algorithm, is recursive. COBWEB classifies instances using a classification strategy similar to the one it uses in learning. However, unlike the learning process, there is no lasting change to the hierarchy because instances are not permanently incorporated as classification takes place. We now describe each step in more detail.

start. Remove an attribute from the instance

COBWEB removes an attribute from the instance, replacing the missing attribute from the previous prediction iteration (if any).

step 1. Preview the next level.

As in learning, COBWEB classifies the instance to the root by default. Then it considers the classes that exist at the next level and calculates category utility for the following partitions:

1. Classification to an existing concept.

   COBWEB tries putting the new instance into each of the existing classes, seeking an existing concept that fits the instance well.

2. Classification as a new concept.

   COBWEB tries setting the instance apart from existing classes, creating a new, disjunct class with the new instance as the only member.

Unlike when learning, COBWEB does not consider merging or splitting nodes during classification.

step 2. Recurse to the next level

The system repeats from Step 1 for the next level down. As COBWEB recurses, it continues to classify the instance to concepts of increasing specificity. COBWEB halts when the instance is placed in a class by itself, becoming a new disjunct, rather than being incorporated into an existing concept at the current level. If the system would create a new disjunct, it predicts from the parent of this disjunct. If the system classifies to a singleton, it predicts from that node.

first instance:
white tennis ball

DIAMETER	2.50
COLOR	0.00
REBOUND	0.70

hierarchy:

P(C_0) = 1.00		
attr	*mean*	σ
DIAM	2.50	0.10
COLOR	0.00	0.10
REBND	0.70	0.10

Figure 7. COBWEB hierarchy after the first instance of a ball.

finish. Predict missing attribute

From the concept node located during classification, COBWEB determines the most frequently occurring value for the attribute that is missing from the instance and it predicts this value. If there are ties, it predicts the first of the tied values listed at the node.

2.5 Sample Execution

In this section we step through a sample execution to illustrate the COBWEB learning algorithm. Most of the example hierarchies used as illustrations thus far have involved nominal attributes. Here, for contrast, we use instances with real-valued attributes. Note that we are using the same domain (game balls), but described numerically. Each instance is described in terms of its diameter (in inches), its color (on a continuous color spectrum numbered from zero to ten, with 0 white and 10 black), and its percent rebound when dropped from a standard height onto a smooth surface. We set the acuity parameter to 0.1 before the run begins. Recall that the acuity parameter specifies the minimum standard deviation, which is required to calculate category utility when COBWEB evaluates partitions that include singleton concepts.

In this case, the system begins with a null concept hierarchy. (Note that alternatively, we could pass a predefined hierarchy into the system, providing a form of background knowledge.) COBWEB reads in the first instance, which is a description of a white tennis ball, and uses it to form the root of the hierarchy. This root concept is illustrated in Figure 2.5. We show concepts numbered in the order in which they are formed, so this is C_0. At the top of the concept description is the probability of its occurrence within the partition created by its parent. Since C_0 is the only node in the hierarchy, $P(C_0) = 1$. Below this concept probability is the list of attributes and a mean and standard deviation for their values, calculated from the instance description. Since this node describes a single instance, the mean values are the same as the values found in the instance. Note that the standard deviations displayed are actually the acuity value, 0.1, since the true standard deviations are zero for this singleton concept.

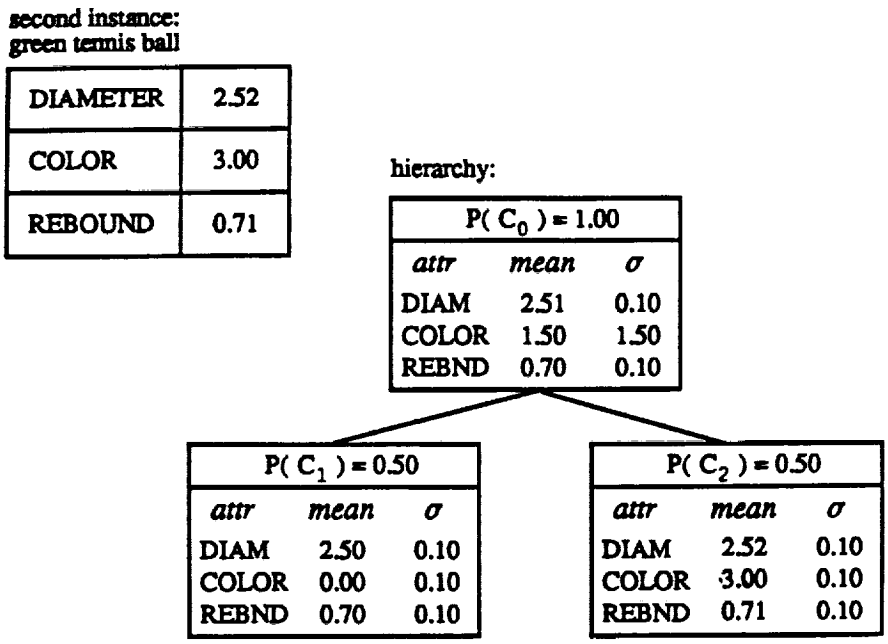

Figure 8. COBWEB hierarchy after the second instance of a ball.

Next, COBWEB reads in a new instance, a description of a green tennis ball. The updated concept hierarchy is shown in Figure 2.5. Using the Incorporate operator, COBWEB incorporates the new instance into the root node, forming a generalized concept for the balls seen thus far. The mean and standard deviation values in this concept summarize the values seen in the two instances. Note that only the standard deviation for the color attribute exceeds the acuity. Since the root was a singleton before the new instance was incorporated, the Incorporate operator extends the hierarchy downward, making the white tennis ball and the green tennis ball instances disjuncts of the new root. Now the two children of the root node describe the individual instances.

The third instance describes a basketball. Figure 2.5 shows the hierarchy updated to include this instance. As always, the instance is first incorporated into the root node. The basketball has quite different characteristics than the two tennis balls and so it changes the means and standard deviations in the root node quite a bit from their previous values. At the next level, the category utility evaluation tells COBWEB to apply the Create-new-disjunct operator, creating a new singleton concept rather than incorporating the basketball into either of the existing tennis ball concepts.

The fourth instance describes a second basketball. Note that the two basketballs are more alike than the two tennis balls, because they are similar not only in diameter and rebound characteristics, but also in their color. Figure 2.5 shows the hierarchy after the new instance has been sorted downward and incorporated. After incorporation into the root node, category utility evaluation determines that the two basketball descriptions are enough alike to be grouped together. COBWEB applies the Incorporate operator at the next level, adding the second basketball to the existing basketball concept node. Since this is a singleton concept, COBWEB extends the hierarchy downward so the two basketball instances become singleton disjuncts.

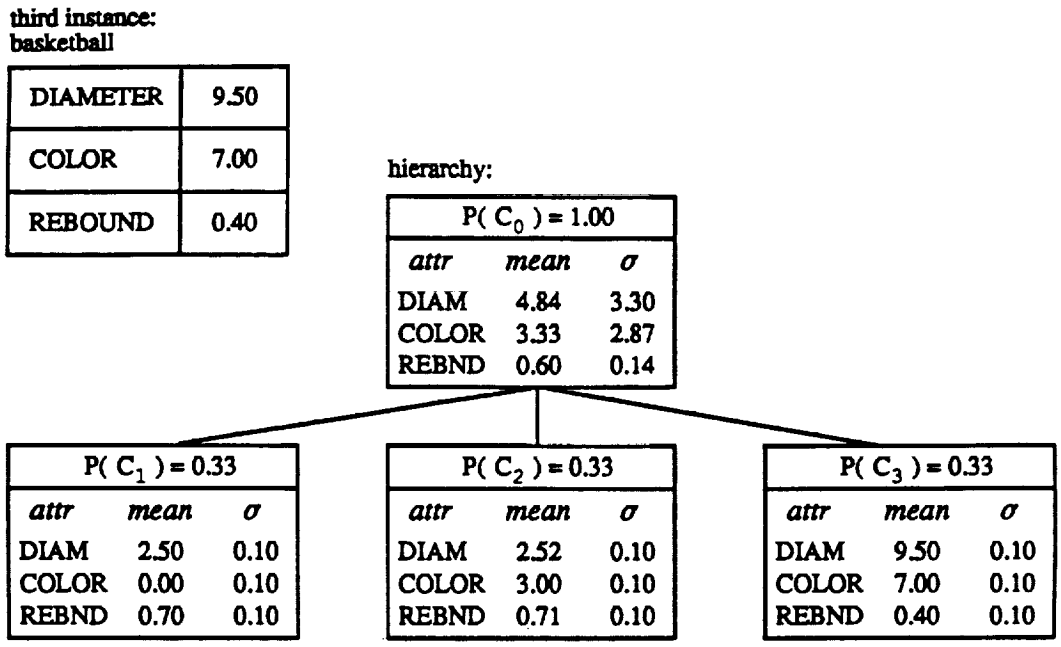

Figure 9. COBWEB hierarchy after the third instance of a ball.

The fifth instance describes a marble. It has very different characteristics from the other four balls: it is much smaller, it does not bounce, and it is the only ball that is red in color. As shown in Figure 2.5, after incorporating the marble description into the root node category utility directs COBWEB to apply the Create-new-disjunct operator rather than classifying the marble with the basketball concept or either of the tennis ball concepts.

The sixth instance describes another green tennis ball. With the incorporation of this instance (Figure 2.5), we see the effects of COBWEB's merge operator. In light of the additional instance descriptions added to the hierarchy since the tennis ball instances came in, these two instances are now relatively similar. The evaluation function directs COBWEB to apply the Merge operator, merging the two tennis ball nodes and incorporating the new tennis ball into this merged node. At the next level down, COBWEB incorporates the green tennis ball into the existing singleton node for the original green tennis ball, creating a new concept node for this subclass. The hierarchy is extended downward, retaining the green tennis ball instances as singleton concepts at the bottom of the hierarchy.

At six instances, COBWEB has already formed some interesting concepts. The root node is too general to contain much information, but below the root we see identifiable concepts. Note the concept for tennis balls, C_7. By studying the means and standard deviations for this concept, we conclude that these balls are about 2.5 inches in diameter, rebound 70% of their dropped height, and are greenish in color. We are less sure of the color because of the high standard deviation of this parameter. Similarly, COBWEB forms a more specialized concept for "green tennis balls", C_2, and a concept for "basketballs," C_3. The system forms these classes on its own; the instances it classifies are not labeled and do not contain any explicit class infomation. If COBWEB were to

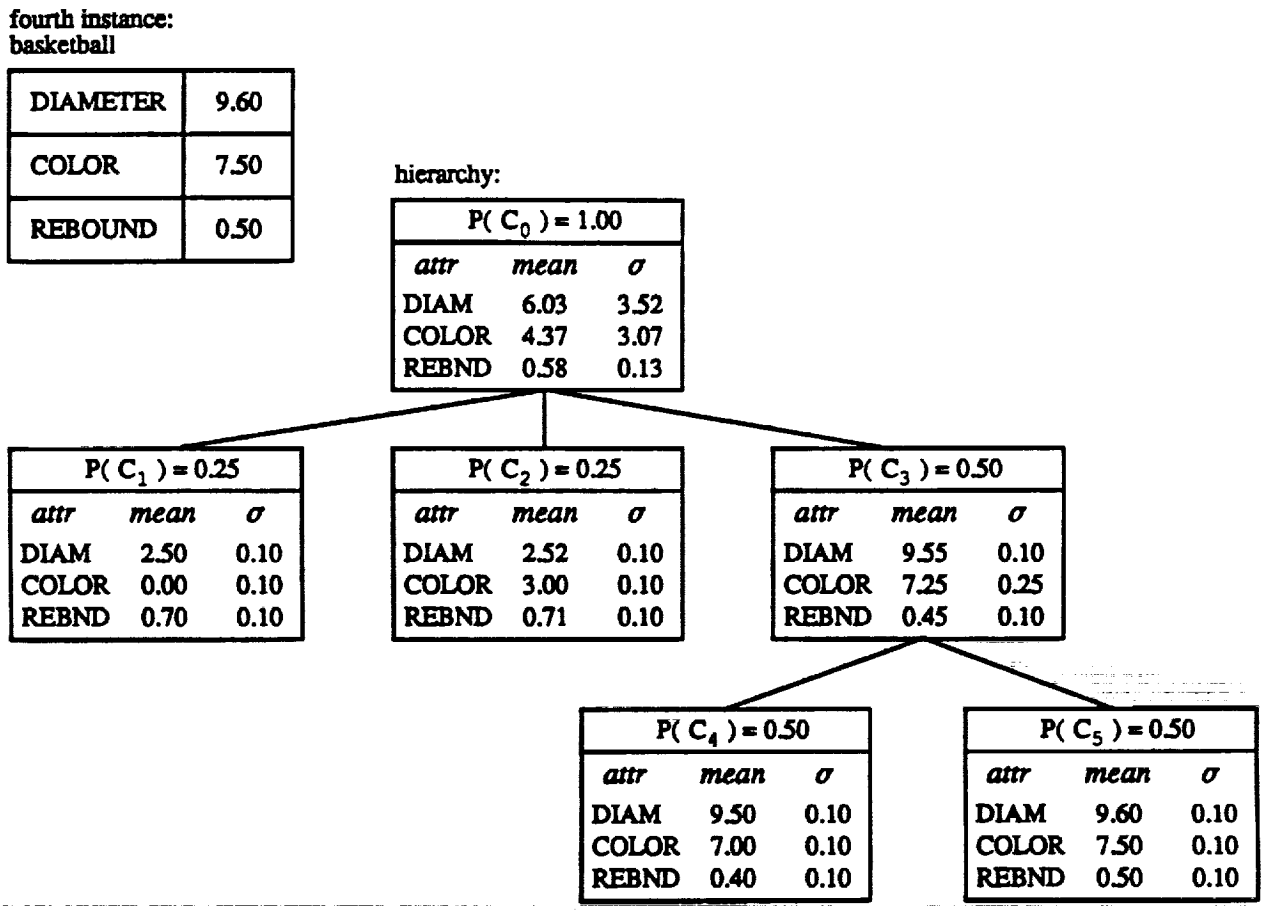

Figure 10. COBWEB hierarchy after the fourth instance of a ball.

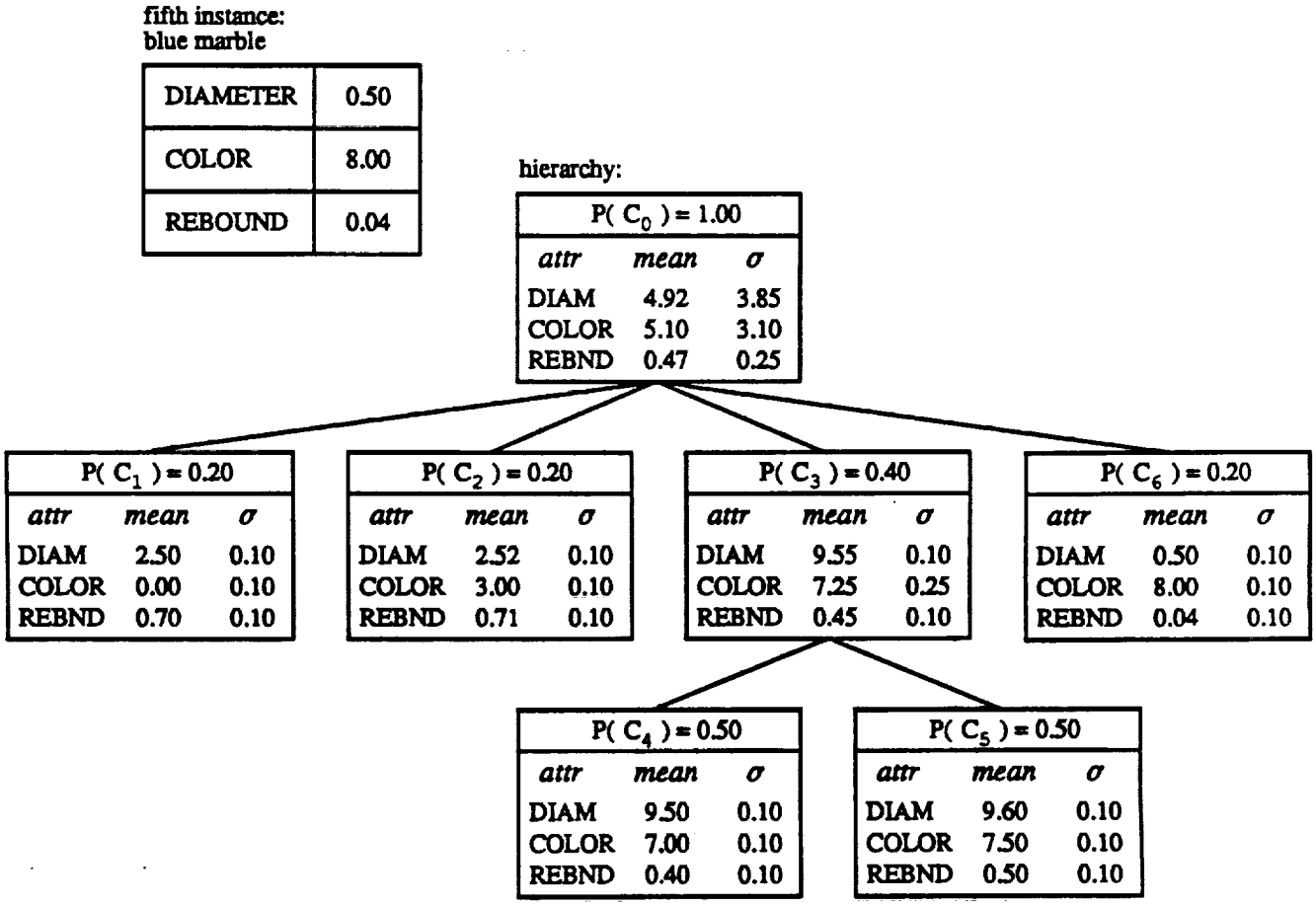

Figure 11. COBWEB hierarchy after the fifth instance of a ball.

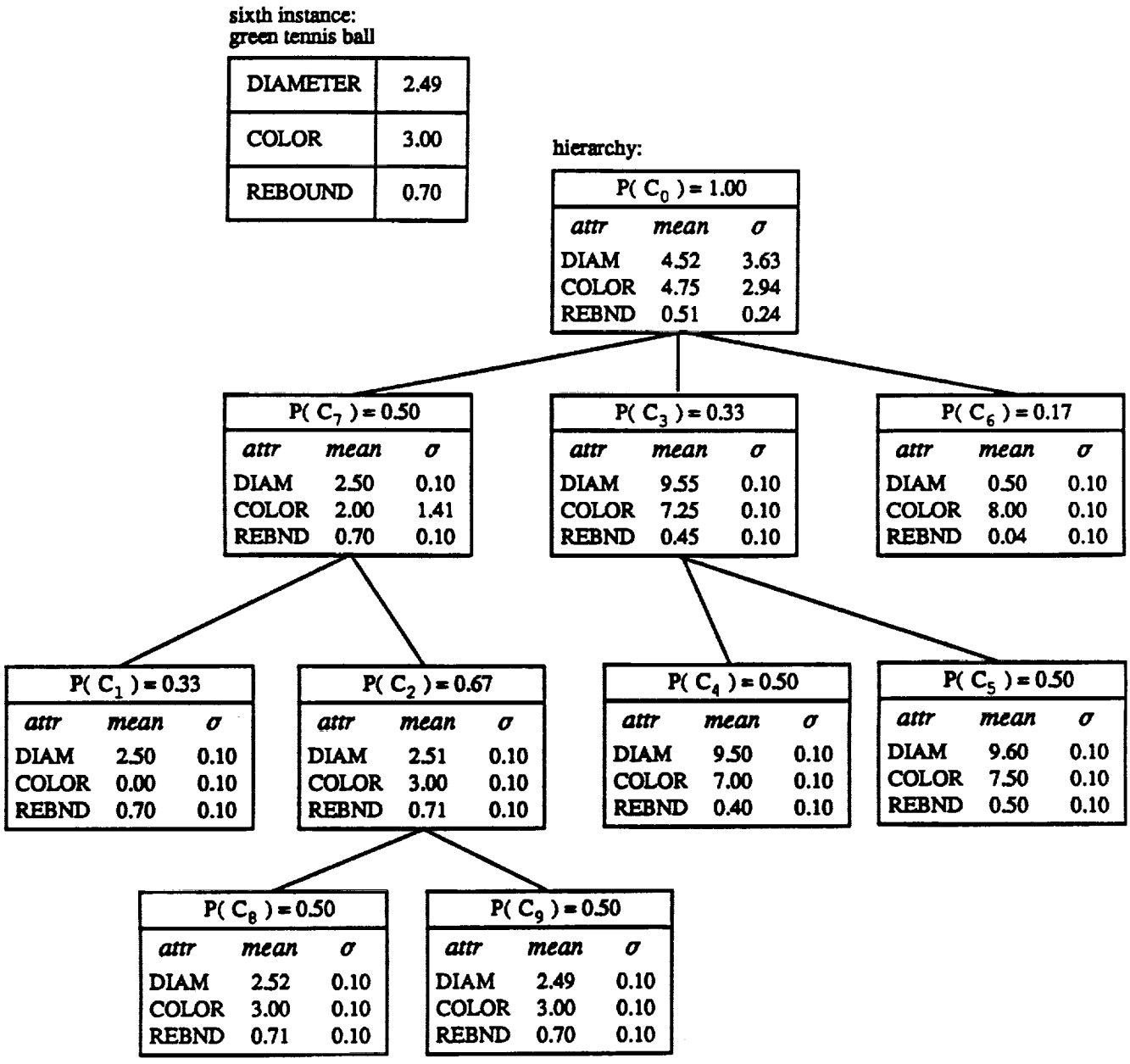

Figure 12. COBWEB hierarchy after the sixth instance of a ball.

process additional instances, it would continue to adjust concept descriptions, form new concepts, and alter the shape of the hierarchy to incorporate the new information.

2.6 Intellectual Debts

As we have mentioned, the COBWEB/3 algorithm described in this document is based on previous work by Fisher (1987a, 1987b), who developed the original COBWEB, and Gennari (1989a, 1990), who developed the CLASSIT system. Fisher's COBWEB formed concept hierarchies from instances with nominal attribute values, using the original (nominal) form of category utility and the four operators described above. Gennari's CLASSIT extended COBWEB by adapting the category utility equation to handle both numeric attributes and instances with mixed nominal and numeric attributes, as described in Section 2.3, including the use of an acuity parameter. The current implementation, COBWEB/3, is more similar to CLASSIT than to the original COBWEB, but we have retained the latter name for the sake of continuity.[3]

However, COBWEB/3 does not incorporate all of the capabilities of its predecessors. For instance, Fisher's system included a 'promotion' operator that moved nodes higher in the concept hierarchy. CLASSIT employed a form of pruning that avoided sorting an instance to the bottom of the hierarchy if it matched well enough against a nonterminal concept. Gennari's system also included a mechanism for matching and incorporating instances with multiple components, as well as a method for selectively attending to diagnostic attributes. Future implementations may draw on these additional capabilities, but COBWEB/3 does not incorporate them.

3. COBWEB/2 was an earlier extension to COBWEB, described in Fisher (1987b).

3. Using the Implementation

COBWEB/3 is an implementation of the COBWEB algorithm that we describe in the preceding sections. COBWEB/3 is not the simplest, most compact, most efficient implementation of COBWEB. More effort has been put into usability and documentation than efficiency. This is research code that includes the LABYRINTH system (Thompson, 1989; Thompson & Langley, in press), an extension to COBWEB that is still under development and undocumented in this release. A more elegant implementation of COBWEB is possible if the constraints of LABYRINTH are absent. Software implementing a version of CLASSIT, a related system we mentioned above, is available at `ics.ics.uci.edu` (directory `pub/classweb`); this code is smaller and probably a bit more efficient. Features in COBWEB/3 not shared by the CLASSIT code include:

1. A graphics interface that allows dynamic inspection of hierarchies and node contents, as well as capability for users to build hierarchies themselves, with guidance from the system evaluation functions. These features are supported in (FRANZ ALLEGRO COMMON LISP) and X windows only;
2. A series of top-level switches to control how nodes are printed, specify how much output to show, pass in pre-defined hierarchies, and request other run-time features;
3. Performance mechanisms for testing the system that allow prediction of missing attributes.

This is our first release to the "outside world", so there may be problems, although this code (at least most of it) has been used for over a year for research purposes. If you have problems or comments, please feel free to send mail to `labyrinth@ptolemy.arc.nasa.gov`, and we will try to get back to you in a short time. In addition, please register with that e-mail address if you are using the code, to keep us informed about who is using the implementation.

3.1 Getting Started

COBWEB/3 was originally implemented using KYOTO COMMON LISP (KCL) and its descendant AKCL from Austin. The current version is written in FRANZ ALLEGRO COMMON LISP, but very little non-standard code is used. Apart from the windowing environment and graphical interface the only significant implementation-dependent code lies within the print functions, which we discuss below. In addition, compilation of this system requires the MIT loop macro; this should be available to most users, but is available with COBWEB/3 if not. The current implementation has been tested on FRANZ ALLEGRO COMMON LISP version 3.1, LUCID COMMON LISP version 3.0, and AKCL version 1.243. We plan to test it under HARLEQUIN LISPWORKS and SYMBOLICS COMMON LISP in the near future.

3.1.1 GETTING THE CODE

COBWEB/3 is available through anonymous FTP from `muir.arc.nasa.gov (128.102.112.24)`, as file `pub/icarus/cobweb.tar.Z`. Make sure you are in "binary" mode when transferring the file,

since it is in UNIX compressed archive format. If you do not have access to UNIX, let us know. Otherwise, execute the UNIX shell command

```
uncompress -c cobweb.tar.Z | tar xf -
```

to get the source from the archive. If you have FRANZ ALLEGRO COMMON LISP with ALLEGRO COMMON WINDOWS on either a Sun 3 or a Sun 4, you can use the graphical interface as documented in Section 3.4. You must get either `code/gr4.fasl.Z` or `code/gr3.fasl.Z` for Sun 4's and 3's respectively. Uncompress the appropriate file and rename it to `gr.fasl` in directory `code`.[4]

3.1.2 INSTALLING COBWEB/3

If you are using a Common Lisp that is not either FRANZ ALLEGRO COMMON LISP, LUCID COMMON LISP, KYOTO COMMON LISP, you need to redefine the functions in `implementation-dep.cl`. COBWEB/3 will function without defining these functions, but much flexibility for memory inspection will be lost. Please send mail to us if you successfully port these functions to a new implementation.

In addition, because Common Lisp lacks a default extension for lisp source files, you might need to change the file type from the default ".cl". This can be done in UNIX by typing:

```
foreach file (*.cl)
? mv $file $file:r.lisp
? end
```

to change all the file types to "lisp".

3.1.3 COMPILING AND LOADING COBWEB/3

COBWEB/3 comes bundled with a system declaration file, `sysdcl.cl`, and a defsystem utility, `defsys.cl`. Although many Common Lisp implementations have their own defsystem utilities, there is not yet a standard, so we use a very simple version here. This system declaration defines the proper order to compile the constituent files. Executing

```
(compile-file "defsys")
(load "sysdcl")                ;; also loads the defsystem utility
(use-package :labyrinth)       ;; or (in-package :labyrinth)
(compile-it)
```

should compile and load the system without errors, though there will be several warnings. For any later uses, executing

4. Note that this is an experimental version of a tree grapher that FRANZ has graciously granted us permission to distribute in binary form for this specific purpose. As it is not yet freely available, please do not distribute it further without permission from FRANZ.

```
(load "sysdcl")              ;; loads the defsystem utility
(use-package :labyrinth)
(load-it)
```

loads the system.

3.2 Input to COBWEB/3

COBWEB/3 receives input from an external data file. The system accepts as input instances described in terms of attributes and values. For an eye these might be

```
color     - blue              hue            - 8.0
shape     - almond     or     width          - 0.74
pupil     - dilated           pupil dilation - 0.80
condition - bloodshot         redness factor - 0.85
```

In general, the more attributes you use the more complete the description. We show attribute labels like color and shape here for clarity only. The system expects no attribute labels, as we explain below.

Table 1. The COBWEB/3 syntax for instances.

$< input-set > ::= < instance > \mid < input-set > < instance >$
$< instance > ::= (< label > < value-list >)$
$< value-list > ::= < value > \mid < value-list > < value >$

3.2.1 COBWEB/3 INPUT FORMAT

Each instance input to COBWEB/3 is in the form of a list. The syntax of a set of input instances is shown in Table 1. The basic syntactic categories are:

- $< label >$, which is a label or Lisp atom.
- $< value >$, which is a nominal or numeric value.

Thus a single instance description would take the form

$(< label > < value > [< value > \ldots])$

For a particular input set, you define the number of attributes and how to label the instances. Note that no attribute labels appear in the instance description. By ordering the attribute values, you designate the attribute each refers to implicitly.

Instances are themselves lists, but each instance stands alone. The input file should be in the form of a number of distinct instances, not one long list. Since the Lisp reader handles input, you

may insert comments anywhere, preceded by a semicolon. The Lisp reader ignores everything on the line following the semicolon.

3.2.2 COBWEB/3 INPUT EXAMPLES

The distribution is bundled with a sample data file, `sample-cob.dat`. This file should be fairly self-explanatory and allow testing of your local installation. Use (run "sample-cob.dat" :force-nominal t), then look at the resulting tree. We illustrate some other examples here.

An input file of eye descriptions described nominally with four attributes might look like this:

```
;; example nominal eye descriptions
(eye-1 blue almond dilated bloodshot)
(eye-2 brown oval normal normal)
(eye-3 blue triangular constricted normal)
(eye-4 hazel oval normal irritated)
```

Here, the implicit attributes are `color`, `shape`, `pupil-condition`, and `redness-factor`. An input file of eye descriptions described numerically with four attributes might look like this:

```
;; example numeric eye descriptions
(eye-1 8.00 0.74 0.80 0.85)
(eye-2 3.50 0.60 0.50 0.10)
(eye-3 8.00 0.69 0.20 0.10)
(eye-4 7.50 0.62 0.52 0.55)
```

For this example, the implicit attributes are `hue`, `width`, `pupil-dilation`, and `redness-factor`.

To specify missing attributes, simply use the value ? in place of the normal attribute value. The value COBWEB/3 recognizes as missing is in the lisp variable ***missing-value***, which can be changed if necessary.

3.3 Top-Level Switches

Issuing a call to the top-level function (run) starts a COBWEB/3 run. This function reads in instances one at a time from an external data file which you specify as a command-line argument. The system processes them, creating a concept hierarchy indexed by the global variable ***isaroot***.[5]

You can control many details of a COBWEB/3 run by setting optional switches when you call (run). Here we describe the switch options and give some examples illustrating their effects.

To initiate a run, designate the input file, and specify switches, type

```
(run "infile" [:switchname setting] [:switchname setting] ...)
```

You can set any combination of switches, although some combinations are more sensible than others. For example, you probably do not want to print out "complete" node information (the **:print-function** switch) when you are running the grapher. We note other dubious interactions below, where applicable, in the explanation of individual switches.

If you want to view a summary of switch settings while COBWEB/3 is loaded, type (**usage**). The function **usage** provides fairly extensive on-line help to the command switches from this manual; from time to time, the on-line help might be slightly more current, let us know if there are discrepancies.

Detailed information about individual switches is on the following pages. In Table 3.3.1 we give an overview of the switches, grouped together by function. Scan the table to get a general idea of how you can use the switches to define COBWEB/3's behavior during a run. Section 3.3.2 gives a detailed description of each switch, organized alphabetically. Refer to the detailed descriptions for in-depth information about switches you are interested in using.

At the end of this section, we provide a summary description of each the switch functions. Use the summary as a reference when running COBWEB/3, after you are familiar with the switches.

3.3.1 OVERVIEW: CLASSES OF SWITCHES

The optional switches control six separate aspects of processing. Table 3.3.1 lists the switches in terms of their functional groupings.

input: You always specify a data file for COBWEB/3 to read in. Optionally, you can also specify an initial concept hierarchy to use as a starting point for classifying incoming instances.

output: You can specify the level of detail and format of the system output.

5. All symbols described in this section are in the **labyrinth** package (and presumably exported). When you issue a **use-package** or **in-package** command, as described in Section 3.1.3 you make these symbols available. If for some reason you are not in the **labyrinth** package, you may need to qualify some symbols defined in the package with the prefix **labyrinth::**. Refer to your Lisp documentation on packages for more information.

Table 2. Summary of COBWEB/3 switches.

Functional Group	Switch Name
INPUT	:att-names :tree
OUTPUT	:outfile :prediction-print :print-each :print-function :printing :too-many-members
INSTANCE PROCESSING	:acuity :force-nominal :merge :split
PERFORMANCE	:pred-atts :prediction :start-at :test-set
DEBUGGING	:breaker :consistency-checks :print-scores
GRAPHER	:build-tree :graph-each :host :keep-this-graph :use-big-window

instance processing: You can specify options for processing incoming instances. These govern how attributes are handled when COBWEB/3 reads in an instance description, and which of the operators to consider applying during classification. In addition, if the input instances have numeric attributes, one can optionally set the acuity.

performance: You can assess the effectiveness of the concept hierarchy by running COBWEB/3 in prediction mode with incoming instances.

debugging: You can request debugging aids.

grapher: You can turn on a graphical display of the developing concept hierarchy and, optionally, participate in classification decisions.

3.3.2 DETAILED SWITCH DESCRIPTIONS

We describe each switch in detail on the following pages. Each switch has a default setting, which is listed first and marked with a [D]. You can change most default values by editing the

(defparameter *default- *variable-name** **<value>)**

definitions in the file **globals.cl**, or using **setq** from the lisp listener. For example, to change the default for the **:acuity** switch, change the variable ***default-acuity***. Most switches are re-set to their defaults with each run. We list the other available settings below the default switch setting.

You can alter switch settings during a run by issuing a break (interrupt) and changing the global variable associated with a switch. Do this through the lisp listener using **setq**, or use the **Parameters** selection on the grapher menu. For more information about the **Parameters** menu, refer to Section 3.4 on using the graphical interface.

For each switch documented, there is a **purpose** section, describing its overall use and function, and a **description** section describing the effect of each of the possible switch settings. If applicable, **examples**, **special notes**, and **bugs** sections give additional information about the switch.

:acuity

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:acuity	1.0 [D]	acuity defaults to 1.0	INSTANCE PROCESSING
	<num>	set acuity to <num>	

Purpose

Use the :acuity switch to set the acuity used by COBWEB/3 for processing continuously-valued attributes. Acuity is a free parameter used in CLASSIT (see Gennari et al, 1989) as a minimum value for σ, the standard deviation seen in the equations of Section 2.3.

Description

If you specify an acuity value, COBWEB/3 uses that value in the current run.

By default, acuity is 1.0.

Special Notes

If the input instances contain no real-values attributes, setting the acuity has no effect.

:att-names

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:att-names	nil [D]	generate standard attribute names	INPUT
	<list of names>	use these names	

Purpose

Because COBWEB/3 uses a compact format for its datasets, you do not specify attribute names in the data files you provide to the system (see Section 3.2). Ordinarily, COBWEB/3 generates attribute names for its internal use of the form

```
(ATT-1 ATT-2 ATT-3 ATT-4 ...)
```

If you plan to inspect the trees COBWEB/3 builds with the **:print-tree** or **:graph-tree** functions, however, you may want to specify more meaningful attribute names.

Description

If you specify a list of attribute names in the form

```
(COLOR SHAPE SIZE ...)
```

then COBWEB/3 uses these names during the current run.

Examples

```
(run "sample.dat" :force-nominal t :att-names '(COLOR SHAPE SIZE))
```

:breaker

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:breaker	nil [D]	no break in main-loop	DEBUGGING
	t	break in main-loop	

Purpose

Use the **:breaker** switch to request a break signal after each instance COBWEB/3 processes, so you can check the execution stack, inspect or change variable values, or execute Lisp expressions. The break signal is embedded in the **(main-loop)** function, which is the main loop of the COBWEB/3 code. The **(main-loop)** function is in the `top.cl` file.

Description

If you specify **:breaker t**, execution breaks in main-loop after each instance is processed.

By default, when **:breaker** is **nil**, execution does not break in **main-loop**.

:build-tree

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:build-tree	nil [D]	no interactive hierarchy building	DEBUGGING
	t	build hierarchy interactively from instances	

Purpose

You may want to provide background knowledge to COBWEB/3 in the form of a concept hierarchy. One way to build such a hierarchy is with the **:build-tree** option.

Description

If you specify **:build-tree t**, you build the concept hierarchy by interacting with the tree grapher as COBWEB/3 runs.

By default, when **:build-tree** is **nil**, COBWEB/3 builds the concept hierarchy on its own, without advice from the user.

Special Notes

Save the hierarchy after building it with the **Save This Tree** option on the Grapher menu. For more information about the grapher, refer to Section 3.4 of this guide.

Pass the hierarchy into COBWEB/3 with the **:tree** switch.

:consistency-checks

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:consistency-checks	nil [D]	no consistency checks	DEBUGGING
	t	perform consistency checks	

Purpose

Using the **:consistency-checks** flag, you can request COBWEB/3 to perform various consistency checks as it executes. These are really program debugging aids, not conceptual debugging aids; their main purpose is to determine if the program is doing anything wrong. They are useful if a bug appears, to track down its source. When a bug is present, COBWEB/3 prints out a relatively useless message if no consistency checks are set. Turning on the consistency check code may help you to flag the bug. These consistency checks can be quite time-consuming, but if no bugs are present, they are silent.

Description

If you specify **:consistency-checks t**, COBWEB/3 will perform various checks during its run, and report any found inconsistencies; otherwise, there will be no visible difference except in execution speed. Among these checks are:

1. checking if counts are consistent, such as whether the **node-count** of a node equals the sum of the **node-counts** of its children;
2. checking consistency of the tree;
3. checking whether pointers between parents and children are consistent.

We do not intend this list to be exhaustive; COBWEB/3 also carries out other checks.

:force-nominal

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:force-nominal	nil [D]	allow primitive attributes to be nominal or continuous	INSTANCE PROCESSING
	t	treats all primitive attributes as nominal	

Purpose

COBWEB/3 can process objects whose attributes are either nominal or numeric. With the **:force-nominal** switch you can force objects with mixed attribute types into having nominal values for all attributes.

Description

If you specify **:force-nominal t**, COBWEB/3 treats all primitive attributes as nominal, even numeric attributes. If all attributes are nominal anyway, this switch has no effect.

By default, when **:force-nominal is nil**, COBWEB/3 checks to see whether the primitive attributes are nominal or numeric, and processes the instance accordingly.

Special Notes

COBWEB/3's evaluation function can handle instance descriptions in which nominal and numeric attributes are mixed, but this approach is still being researched. If you have mixed attributes, an alternative option is to set **:force-nominal t** or to seek a way to represent the nominal attributes numerically.

:graph-each

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:graph-each	nil [D]	no graphing	GRAPHER
	t	show graph after each instance	

Purpose

COBWEB/3 can display its concept hierarchy graphically, in addition to the printed output it provides. Use the **:graph-each** switch to request graphical output.

Description

If you specify **:graph-each** t, COBWEB/3 displays the concept hierarchy as a graph. A new graph displays after COBWEB/3 incorporates each new instance.

By default, when **:graph-each** is **nil**, COBWEB/3 does not display the concept hierarchy graphically. The system generates only printed output as it runs.

Special Notes

For more information about the grapher, refer to Section 3.4 of this guide.

:host

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:host	(machine running Lisp) [D] < *host* >	use X display running Lisp draw graphics on given host	GRAPHER

Purpose

COBWEB/3 uses an X11-based graphics package. In X, a given process can display its output on other displays (given permission, see "xhost" for details). By default, if graphics are called for, COBWEB/3 will draw them to the main display of the CPU on which Lisp resides. However, specifying an alternative machine with the :host option allows alternative displays.

Description

To specify an alternative display for graphics, supply a string with the name of that machine.

Examples

```
(run "sample-cob.dat" :force-nominal t :graph-each t :host "thoreau")
```

will display the graphics on host "thoreau".

:keep-this-graph

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:keep-this-graph	nil [D]	redraw developing hierarchy in one window	GRAPHER
	t	create new window for each new hierarchy	

Purpose

Ordinarily when the grapher is running, COBWEB/3 displays the concept hierarchy in a single window, overwriting the previous hierarchy with each new one as instances are incorporated. If you prefer, you can request separate windows for each new graph with the **:keep-this-graph** switch.

Description

If you specify **:keep-this-graph t**, a new graph window displays every time COBWEB/3 updates the graph of the concept hierarchy.

By default, when **:keep-this-graph** is **nil**, COBWEB/3 redraws the updated graph in a single window, overwriting the previous graph.

Special Notes

You can change the value of **:keep-this-graph** in the middle of a run using the **Parameters** window, which is explained in Section 3.4 of this guide. By turning **:keep-this-graph** on and off you can selectively keep graphs of hierarchies you find interesting or want to study further.

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:merge	t [D]	allow merge operator	INSTANCE PROCESSING
	nil	disallow merge operator	

Purpose

COBWEB/3 uses the merge operator to help it recover from order effects in its input. If you don't want the system to use the merge operator, you can turn merging off with the **:merge** switch.

Description

If you specify **:merge nil**, COBWEB/3 does not consider applying the merge operator as it sorts an instance thorough the hierarchy.

By default, when **:merge** is t, COBWEB/3 considers merging the two best nodes[6] into a single node and incorporating the instance into this newly-created node as it sorts an instance through the concept hierarchy.

6. "Best" node means that when COBWEB/3 incorporates the instance into this node, the resulting partition receives the highest category utility score.

:outfile

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:outfile	nil [D]	output to *STANDARD-OUTPUT*	OUTPUT
	t	output to < *infile* >-out	
	< *outfile* >	output to < *outfile* >	

Purpose

You can specify where you want COBWEB/3's output text to go.

Description

If you specify **:outfile t**, COBWEB/3 sends output to the file "< *infile* >-out", where < *infile* > is the input file name you specified on the command line after run.

If you specify an outfile name, COBWEB/3 sends output to the file "< *outfile* >". Make sure you put the outfile name in quotes.

By default, when **:outfile** is **nil**, the default output of COBWEB/3 messages and data is to ***standard-output***, a Lisp variable. Typically, this is the screen.

Examples

```
(run "foo.dat"...)
⇒ output to *STANDARD-OUTPUT*
(run "foo.dat" :outfile t ...)
⇒ output to "foo.dat-out"
(run "foo.dat" :outfile "outfile" ...)
⇒ output to "outfile"
```

:pred-atts

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:pred-atts	nil [D]	predict with each attribute in turn	PERFORMANCE
	<list of positions>	predict with positionally-specified attributes only	
	<list of names>	predict with named attributes only	

Purpose

In any of the prediction modes, certain attributes are excised from the instance description, one at a time, and the remaining partial instance description is used to predict what the missing value should be. By default, every attribute in the instance description is excised, one at a time. Alternately, you can specify a particular subset of the attributes to be excised and predicted, since not all attributes might be equally interesting or predictable. Thus, one can specify certain attributes to be predicted.

Description

You can specify which attributes to predict in one of two ways: positionally or by name.

To specify positionally, list the positions of the attributes to predict, starting at 0.

To specify by name, list the names of the attributes to predict.

Examples

```
(run "foo.dat" :prediction :test-train :pred-atts '(0 1 3 5))
(run "foo.dat" :prediction :test-train :att-names '(color shape size weight)
               :pred-atts '(color size))
```

Special Notes

Specifying attributes by name using the **:pred-atts** switch works only if you also indicate attributes names using the **:att-names** switch.

:prediction

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:prediction	nil [D]	no prediction	PERFORMANCE
	:test-train	use each training instance as a test instance	
	:test-set-during	use test set between each training instance	
	:test-set-after	use test set after all training	

Purpose

You can test the system's ability to predict each attribute in the instance descriptions. Use the **:prediction** switch to turn on one of three specified prediction modes.

Description

If you specify **:prediction :test-train**, COBWEB/3 uses each instance as a test instance before incorporating it into the hierarchy, thus obviating the need for a separate test set. Thus for each instance, multiple tests are performed. The system excises each of the attributes (see **:pred-atts**), one at a time, and reports a score for that prediction.

If you specify either **:test-set-during** or **:test-set-after**, then a separate test set of instances is used for testing the concept hierarchy (see **:test-set**). In the **:test-set-during** case, the test set is used between every instance (after **:start-at**); this is time-consuming but allows generation of learning curves. In the **:test-set-after** case, the test instances are used only after all learning has taken place, giving only an idea of final predictive performance but taking far less time.

By default, when **:prediction is nil**, COBWEB/3 does not do prediction.

Special Notes

When the prediction mode is **:test-train**, prediction scores are reported as lists in the form

```
(<instance-num> <att-name> <score>),
```

For example, (5 ATT-3 1.0) means that, in predicting the value of the third attribute of the fifth instance (after four instances have been classified), the prediction score is 1.0. If **:outfile** is t, the prediction scores are written out to the file < *outfile* >-rep.

When the prediction mode requires use of a test set, prediction scores are reported as lists in the form

```
(<instance-num> <test-instance-name> <att-name> <score>),
```

since there are multiple test instances in this case.

For nominal attributes, a "prediction score" is simply 0 or 1, depending on whether COBWEB/3 predicted the same value as the one excised from the object. See Fisher (1987a,1987b) for examples of prediction results using these scores.

For numeric attributes, a "prediction score" represents the absolute error between the predicted and real value, a number between 0 and ∞. See Gennari, Langley, and Fisher (1989) or Gennari (1990) for examples of prediction results using these scores. Note that the combination of prediction results for nominal and numeric attributes is thus ill-defined in the current release.

The prediction code was developed for specific research purposes. The implementors welcome suggestions on other approaches to evaluating COBWEB/3's predictive ability.

:prediction-print

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:prediction-print	nil [D]	no trace info during prediction	OUTPUT
	t	print trace info during prediction	

Purpose

When testing the system's predictive performance, you can either have COBWEB/3 print out descriptive information about each prediction it attempts, or simply print the outcome of each of the prediction trials.

Description

If you specify **:prediction-print** t, COBWEB/3 prints out trace information as it does prediction. It prints out information about each prediction and about each classification; this can be informative but time-consuming.

By default, when **:prediction-print** is **nil**, the system does not print out trace information as it does prediction; rather it prints only the outcome of each of the predictions trials.

Special Notes

If the **:prediction** switch is **nil**, setting **:prediction-print** to t has no effect.

:print-each

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:print-each	nil [D]	no tree printout	OUTPUT
	t	print tree after each instance	

Purpose

You can print out the concept hierarchy in text form during a run. You may find this useful if you are not using the graphics and want to view the hierarchy as it develops.

Description

If you specify **:print-each** t, COBWEB/3 prints out the current concept hierarchy after each instance has been classified.

By default, the system does not print out the hierarchy at all during a run.

Special Notes

This switch always uses the **rec-members** print function. To see a different level of detail, you can print out ***isaroot*** with a different recursive print function. Refer to the **:print-function** documentation for more information.

:print-function

SWITCH	OPTIONS	SYNOPSIS	SWITCH
:print-function	:members-compress [D]	node labels from input, compresses	CLASS
	:members	node labels from input, never compresses	OUTPUT
	:generic	generic node labeling, N1, N2...	
	:complete	detailed node information	
	:rec-members-compress	like members-compress, but recursive	
	:rec-members	like members, but recursive	
	:rec-generic	like generic, but recursive	
	:rec-complete	like complete, but recursive	

Purpose

COBWEB/3 makes several print functions available. Set the print function to control how much information to print when you type a node name. There are two classes of print functions: *regular* and *recursive*. Regular print functions print only information about a particular node. Recursive print functions print information about a node, its children, and so on recursively, thereby printing the entire subtree rooted at a given node. Recursive printers are those with a **rec-** prefix.

Description

If you specify **:print-function :members-compress**, COBWEB/3 draws nodes using the labels given by the input instance. When the system creates a generalized node from two or more children, it labels this node with a name that is the concatenation of the labels of the child nodes. When the number of elements in the concatenated label reaches the size designated by the variable ***too-many-members*** (or the **:too-many-members** switch), it displays the number of elements followed by the word **instances**. For example, in lieu of **(ball1 ball2 ball3 ball4 ball5)** it shows **(5 instances)** if ***too-many-members*** is set to 4. We refer to this as *compressed* form.

The **:members** print-function is just like the **:members-compress** print-function, but never compresses.

If you specify **:print-function :generic**, COBWEB/3 labels the nodes with a generic numbering system: N1 for node 1, N2 for node 2, and so forth. Leaf nodes retain the labels given in the input.

If you specify **:print-function :complete**, COBWEB/3 prints detailed node information, including attributes and probabilities on each value. Note that the values are sorted.

The **:rec-members-compress** print-function is just like **:members-compress**, but recursive.

The **:rec-members** print-function is just like **:members**, but recursive.

The **:rec-generic** print-function is just like **:generic**, but recursive.

The **:rec-complete** print-function is just like **:complete**, but recursive.

Special Notes

You can set the print function in two different ways. You can change from the default by setting one of the above keywords when you invoke a run. Alternatively, if you are using an implementation that so allows, you can simply type in a one-word command for each of these switches to a lisp listener (from a break loop or after a run) to change the current default. These macros are named from the keywords given above, without the beginning colon. Thus typing **(members)** changes the current default print function to **:members**.

In some cases, the system ignores the current default print function. For instance, if the **:print-each** flag is on, the **rec-members** function is called by default. Some print functions do not work well with the FRANZ tree grapher, which has severe problems with recursive print functions, and works best with **:members-compress**, **:members**, or **:generic** print functions. The grapher does not work well with the recursive functions nor the **:complete** function. However, the function **(graph-tree)** will use the current default printer if it is a "legal" one for graph use; otherwise, it uses the **members-compress** print function.

The implementors have had great difficulties in understanding how to make all Lisp implementations do useful indentation in all cases with nodes. Suggestions from talented hackers are always welcome.

The variable***isaroot*** in the examples below is the name of the root node of the hierarchy.

Examples

These examples show how each of the nonrecursive printers would print the same hierarchy, in this case the one produced by running COBWEB/3 on the first eight instances of **sample-cob.dat** (the soybean disease data that is supplied with the distribution).

```
;;assuming :members-compress is the current default.
*isaroot* ⇒
D4 D3 D2 D1 D4 D3 D2 D1
(members-tight-printer *isaroot*) ⇒
(8 instances) ;; If *TOO-MANY-MEMBERS* is less than 8
(generic-printer *isaroot*) ⇒
#2
(members-printer *isaroot*) ⇒
#{D4 D3 D2 D1 D4 D3 D2 D1}
```

```
(complete-printer *isaroot*) ⇒
Node{(D4 D3 D2 D1 D4 D3 D2 D1)
   Attribute{ATT-1    count  8
      0               0.25
      2               0.25
      4               0.25
      5               0.125
      6               0.125}
   Attribute{ATT-2    count  8
      1               0.5
      0               0.5}
   Attribute{ATT-3    count  8
      2               0.625
      1               0.125
      0               0.25}
   Attribute{ATT-4    count  8
      1               0.625
      0               0.25
      2               0.125}
   Attribute{ATT-5    count  8
      0               0.625
      1               0.375}
   Attribute{ATT-6    count  8
      1               0.375
      3               0.375
      2               0.125
      0               0.125}
   Attribute{ATT-7    count  8
      1               0.625
      3               0.125
      2               0.125
      0               0.125}
   Attribute{ATT-8    count  8
      1               0.75
      2               0.25}
   Attribute{ATT-9    count  8
      0               0.5
      1               0.5}
      .
      .
      }
```

:print-scores

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:print-scores	0 [D]	no score printing	DEBUGGING
	1	information about four classification operators	

Purpose

COBWEB/3 can print out some of the scores it uses in its internal evaluation of which operators to apply. This is useful for seeing how the algorithm is working in a given domain.

Description

There are two different settings for the **:print-scores** parameter. With its default value of 0, no information about the scores of possible operator applications is printed. With a value of 1, the system prints each score of whether to apply the new-disjunct, incorporate, merge, or split operator to a partition.

:printing

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:printing	t [D]	output to designated :outfile	OUTPUT
	nil	no output	

Purpose

You can suppress the printed output generated by COBWEB/3. Because printing output takes up a lot of time during a run, we recommend turning output off during full-scale test runs to improve speed.

Description

If you specify **:printing nil**, COBWEB/3 generates NO text output during a run.

By default, when **:printing** is t, COBWEB/3 sends output to the **:output** location. Output in general (depending on other switch settings) consists of lines indicating what instance is being processed (indicated by its label), and whether the system is in learning, prediction, or recognition mode. For each classification, COBWEB/3 reports what operator is applied at each level. The system also reports the run time in minutes, to facilitate monitoring of experiments.

:split

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:split	t [D]	allow split operator	INSTANCE PROCESSING
	nil	disallow split operator	

Purpose

COBWEB/3 uses the split operator to help it recover from order effects in its input. If you do not want the system to use the split operator, you can turn splitting off with the :split switch.

Description

If you specify :split nil, COBWEB/3 does not consider applying the split operator as it sorts an instance through the hierarchy.

By default, when :split is t, the system considers splitting the best node[7] (moving the best node's children up a level) and incorporating the instance into the best of the children.

7. "Best" node means that if COBWEB/3 incorporates the instance into this node, the resulting partition receives the highest Category Utility score.

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:start-at	2 [D]	start prediction at 2nd instance	PERFORMANCE
	< *n* >	start prediction at nth instance	

Purpose

Use the **:start-at** switch to tell COBWEB/3 at what input instance to begin prediction.

Description

You can specify an integer corresponding to the instance at which prediction should start.

By default, prediction starts at the second instance (when **:prediction** is **:test-train** or **test-set-during**).

Special Notes

If **:prediction** is **nil**, setting the **:start-at** value has no effect.

:test-set

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:test-set	< *infile* >-test [D] < *file* >	generate testset file name from < *infile* > use < *file* > for test data	PERFORMANCE

Purpose

Use this switch to indicate the file containing the test set, when **:prediction** is **:test-set-during** or **:test-set-after**.

Description

If you specify a test file name, COBWEB/3 reads the test set from the file "< *file* >". Make sure you put the test file name in quotes.

By default, COBWEB/3 reads the test set from the file "< *infile* >-test, where < *infile* > is the input file name for the training instances.

Examples

```
(run "foo.dat" :prediction :test-set-after)
⇒ look for test set in "foo.dat-test"
(run "foo.dat" :prediction :test-set-after :test-set "big-test")
⇒ look for test set in "big-test"
```

Special Notes

If **:prediction** is **nil** or **:prediction**, setting **:test-set** has no effect.

:too-many-members

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:too-many-members	10 [D]	compress with 10 objects	OUTPUT
	< *n* >	compress with *n* objects	

Purpose

Use the **:too-many-members** switch to tell COBWEB/3 at what point to begin printing compressed node labels. When a node label describes more than the specified number of objects, a compressed label prints instead of using the concatenation of all the instance names. This switch works in conjunction with the **:print-function** switch.

Description

You can specify an integer corresponding to the level at which compression mode starts.

By default, compression starts at ten instances.

Special Notes

This switch operates when **:print-function** is **:members-compress** or **:rec-members-compress**. Wehn other print functions are active, setting **:too-many-members** has no effect.

:tree

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:tree	nil [D]	no input tree	INPUT
	< *lispexpr* >	evaluated expr is root node of input tree	

Purpose

You can give COBWEB/3 background knowledge in the form of a concept hierarchy. Use the **:tree** switch to pass the predefined tree to COBWEB/3 at the start of a run.

Description

If you specify a < *lispexpr* >, the system evaluates the expression, which should evaluate to the root node of the concept hierarchy you want to pass in. By specifying **:tree *saved-tree***, you pass in the most recent tree you saved with the **Save This Tree** option on the grapher menu. You can copy trees with the COBWEB/3 function **(copy-isa-tree)**. Since the system alters the concept hierarchy destructively, use **(copy-isa-tree)** to pass in the same tree for more than one run.

By default, when **:tree** is **nil**, COBWEB/3 begins execution with a null hierarchy.

Examples

```
(run "foo.dat" :tree *saved-tree* ...)
;; *saved-tree* is altered during run
(run "foo.dat" :tree (copy-isa-tree *saved-tree*))
;; *saved-tree* stays intact, a copy is altered during run
```

Special Notes

You can build trees interactively with the **:build-tree** option. Refer to Section 3.4 of this manual for more information about the grapher.

Refer to the definition of **node** in **struct.cl** to find out about the structure of nodes and how they link to form trees.

:use-big-window

SWITCH	OPTIONS	SYNOPSIS	SWITCH CLASS
:use-big-window	nil [D]	no background window	GRAPHER
	t	background window for graphs	

Purpose

Use the **:use-big-window** switch to display a large background window behind the window displaying the concept hierarchy.

Description

If you specify **:use-big-window** t, a large background window displays behind the window displaying the concept hierarchy. This window is useful if you execute window dumps in X. The concept hierarchy and associated windows are children of the background window, so you can dump all these windows at once to the printer.

By default, when **:use-big-window** is **nil**, no background window displays.

Special Notes

When the switch **:graph-each** is **nil**, setting **:use-big-window** to t has no effect.

3.3.3 SUMMARY SWITCH DESCRIPTIONS

This section summarizes the options for each switch. We recommend using these pages as a quick reference for the switches after you are familiar with them.

INPUT

Optional parameters related to INPUT:

SWITCH	OPTIONS	SYNOPSIS
:att-names	nil [D]	generate standard attribute names
	(name-1 name-2 ....)	use these names
:tree	nil [D]	no input tree
	< *lispexpr* >	evaluated expr is root node of input tree

OUTPUT

Optional parameters related to OUTPUT:

SWITCH	OPTIONS	SYNOPSIS
:outfile	nil [D]	output to *STANDARD-OUTPUT*
	t	output to < *infile* >-out
	< *outfile* >	output to < *outfile* >
:prediction-print	nil [D]	no trace info during prediction
	t	print trace info during prediction
:print-each	nil [D]	no tree printout
	t	print tree after each instance
:print-function	:members-compress [D]	node labels from input, compresses
	:members	node labels from input, never compresses
	:generic	generic node labeling, N1, N2...
	:complete	detailed node information
	:rec-members-compress	like members-compress, but recursive
	:rec-members	like members, but recursive
	:rec-generic	like generic, but recursive
	:rec-complete	like complete, but recursive
:printing	t [D]	output to designated :outfile
	nil	no output
:too-many-members	10 [D]	compress with 10 objects
	< *n* >	compress with *n* objects

INSTANCE PROCESSING

Optional parameters related to INSTANCE PROCESSING:

SWITCH	OPTIONS	SYNOPSIS
:acuity	1.0 [D] <num>	use specified acuity set acuity to <num>
:force-nominal	nil [D] t	nominal and/or continuous attributes treats all attributes as nominal
:merge	t [D] nil	allow merge operator disallow merge operator
:split	t [D] nil	allow split operator disallow split operator

PERFORMANCE

Optional parameters related to PERFORMANCE:

SWITCH	OPTIONS	SYNOPSIS
:pred-atts	nil [D] (0 1 3 ...) (color shape)	predict with each attribute in turn predict with positionally-specified attributes only predict with named attributes only
:prediction	nil [D] t	no prediction excise and predict instance attributes
:start-at	2 [D] < n >	start prediction at 2nd instance start prediction at nth instance
:test-set	< *infile* >-test [D] < *file* >	generate testset file name from < *infile* > use < *file* > for test data

DEBUGGING

Optional parameters related to DEBUGGING:

SWITCH	OPTIONS	SYNOPSIS
:breaker	nil [D]	no break in main-loop
	t	break in main-loop
:consistency-checks	nil [D]	no consistency checks
	t	perform consistency checks
:print-scores	0 [D]	no score printing
	1	information about four classification operators

GRAPHER

Optional parameters related to GRAPHER:

SWITCH	OPTIONS	SYNOPSIS
:build-tree	nil [D]	no interactive hierarchy building
	t	build hierarchy interactively from instances
:graph-each	nil [D]	no graphing
	t	show graph after each instance
:host	(machine running Lisp) [D]	use X display running Lisp
	< *host* >	draw graphics on given host
:keep-this-graph	nil [D]	redraw developing hierarchy in one window
	t	create new window for each new hierarchy
:use-big-window	nil [D]	no background window
	t	background window for graphs

3.3.4 USEFUL TOP-LEVEL FUNCTIONS

There are several COBWEB/3 functions which do useful work when you call them from the top level. You can call these functions from the Lisp listener between runs, or during a run after a break executes.

Function
graph-tree

Usage
graph-tree [:node NODE] [:title TITLE]

Description
graph the current tree, or a sub-tree

[Note: This function is applicable only when using the FRANZ tree-graphing package.]

It is often useful to graph the current tree from a break loop after a run is completed. This function takes the current tree rooted at ***isaroot*** and shows it graphically, using **generic-printer** if that is the current default node printer, otherwise using **members-compress-printer**. The user can specify a NODE as the root of the grapher tree. Any parameter TITLE passed in will be used as the window-title for the resulting graph.

The resulting graph will have the following mouse sensitivities:

LEFT produces a full display of that node's internals, in a separate window.

MIDDLE sets the global variable **g*** to that node.

RIGHT gives a partial display of that node's internals.

The function **graph-tree** returns the graph (an internal data structure) and window associated with the display as multiple values.

Function
print-tree

Usage
print-tree [:primitive PRIMITIVE] [:composite COMPOSITE]

Description
graph the current tree, or a sub-tree

This function prints out the current tree to the terminal by calling **rec-members-printer** on the current tree.

Function
initialize-window-system

Usage
initialize-window-system [:host HOST]

Description
Initialize the window system on host HOST.

This function is largely a front-end to the FRANZ function **cw:initialize-common-windows**. However, it also sets certain global variables for the proper functioning of the grapher.

You can call this function directly to request a display HOST other than the default. Specify the name of the host with the :host switch. COBWEB/3 calls this function automatically, if necessary, when the user specifies the **:graph-each** or **:build-tree** switches, or if the user calls **graph-tree**.

Function
write-tree-to-file

Usage
(write-tree-to-file FILE)

Description
write out the concept hierarchy to a file in machine-readable form

This function prints out a concept hierarchy in machine-readable Lisp forms, so that another Lisp process can read in the hierarchy. It overwrites any existing tree in the file designated. As currently implemented, write-tree-to-file is quite slow.

Function
read-tree-from-file

Usage
(read-tree-from-file FILE)

Description
returns a pointer to the tree contained in FILE

A complement to the function **write-tree-to-file** described above, **read-tree-from-file** simply returns the tree contained in FILE.

3.4 Using the Graphical Interface

You can view COBWEB/3's developing concept hierarchy graphically[8] by setting the **:graph-each** switch at run time. When the grapher is running, the system displays a hierarchy after each instance processes. This enables you to view the structure of the hierarchy and examine individual concept nodes.

Set the **:graph-each** switch just as you would any other run time switch. For example, to run the system on the input file sample-cob.dat with the grapher on, issue the command

```
(run "sample-cob.dat" :graph-each t)
```

As the run begins, COBWEB/3 graphs the first input instance and displays a menu.

3.4.1 THE GRAPHER MENU

Using the onscreen menu, you can interact with the system as it processes successive input instances. The menu has these selections:

```
Graph Next Object
Save This Tree
Flush Guts
Parameters
Exit
```

Two selections, `Graph Next Object` and `Exit` directly control the execution of COBWEB/3. The other selections are for saving trees, and for window and parameter maintenance.

Graph Next Object

With the `Graph Next Object` menu selection you control COBWEB/3's processing of instances in the input file. The system processes the first input instance automatically. To process the next instance and redisplay the hierarchy, select Graph Next Object on the grapher menu.

After you select this option, "no-op...graphing" displays on the grapher menu. After the next object processes, which may take several seconds, the current grapher frame clears and the new hierarchy displays. To continue, select Graph Next Object again.

Parameters

Using the `Parameters` selection you can display the Parameters menu, with which you can change the switch settings interactively while COBWEB/3 is running. Once you select Parameters, be patient! The current implementation of the Parameters window is kludgy and the window may take 30 seconds or more to display.

8. If you use Allegro Common Windows, you can use the grapher. Otherwise only nongraphical output is available.

When the window displays, use the LEFT mouse to select the value you want to change. When possible values are T/NIL, each mouse click toggles between these values. When other values are possible, click the mouse to step through the possibilities.

Use the MIDDLE mouse to restore a default value. Selecting a value with the middle mouse brings up a small menu. Select **restore default** from this menu to restore the default value. The default values used are the ones defined in the file **globals.cl** and marked with the comment ";;default settings for switches". (You can change defaults by editing the **defparameter** settings in the **globals.cl** file.)

When you finish selecting parameters, flush the window. If you select Parameters when a Parameters window already exists, you will get a Common Windows error from which it is difficult to recover.

Note that changing parameters between runs has no lasting effect. Instead, set whatever switches you want to at the beginning of a run, and only use the Parameters window if you want to change these values after the run begins.

Save This Tree

To save the data structure for the concept hierarchy that is currently onscreen, select Save This Tree on the grapher menu. COBWEB/3 stores the root of the hierarchy in ***saved-tree***. You can pass this tree into later runs of the system using the **:tree** switch. Each time you select **Save This Tree**, COBWEB/3 overwrites ***saved-tree*** with the current root.

Flush Guts

As we describe in detail in the next section, you can click on nodes in the hierarchy to open a window with information about the node. These windows stay onscreen until you flush them. To flush all the windows containing node information after you have done a node expansion, select Flush Guts on the grapher menu. You can also flush node windows one at a time, as you would any window, by selecting flush on the window's right button menu.

Exit

To stop processing input and exit the run, select Exit on the grapher menu.

3.4.2 GRAPHICAL DISPLAY

The concept hierarchy displays horizontally on your screen. You can control the display format of the nodes and you can request detailed node information.

Node Display

Nodes in the hierarchy are labeled according to the print function in force when COBWEB/3 generates the graph. By default, the print function is **members-tight**. With this print function, the system draws nodes using the labels given by the input instance. When COBWEB/3 creates a generalized node from two or more children, it labels this node with a name that is the concatenation

of the labels of the child nodes. For more information on this and other print functions, see the description of the **:print-function** switch in Section 3.3 of the manual.

To select a different print function, specify the one you want with the **:print-function** switch. You can change the print function during a run using the Parameters menu. This changes only future graphs, not the current one.

Using the Mouse on Nodes

The graphical display shows the structure of the concept hierarchy, but no information about individual nodes. To see information about the frequencies associated with the attribute values at a node, click on the node with either the left or right mouse button.

If you click with the LEFT mouse button, COBWEB/3 provides complete node information. For each attribute represented in the node, you see a list of all the values that the attribute has taken on and the associated conditional probability for that value.

If you click with the RIGHT mouse button, the system shows abbreviated node information. For each attribute represented in the node, you see only the value with the highest associated conditional probability. If the values are numeric, the right mouse button has the same effect as the left button.

After you expand a node, the window containing node information stays onscreen until you flush it. You can flush nodes with the Flush Guts option on the static menu. For more information, review the description of the Flush Guts menu option in this section.

The MIDDLE mouse button can be used as a quick way to get access to a node in the tree from the lisp listener. Clicking this button on a node in a displayed tree sets the global variable **g*** to that node in the concept hierarchy.

3.4.3 MULTIPLE GRAPHS

Ordinarily COBWEB/3 redraws each new hierarchy in the same window, overwriting the previous hierarchy. Alternatively, you can plot successive hierarchies in different windows so that each graph remains onscreen until you flush it.

You can keep successive graphs by setting the **:keep-this-graph** switch to t. Refer to the documentation for the **:keep-this-graph** switch for detailed information. You can also change the value during a run using the Parameters menu.

3.4.4 THE BUILD-TREE OPTION

Instead of allowing COBWEB/3 to build a hierarchy using category utility to evaluate partitions, you can build the hierarchy yourself using the graphical interface as a tool. To run the tree builder, set the **:build-tree** switch to t at run-time. When the **:build-tree** switch is set, the system displays each object from the input file next to the current hierarchy. Using the mouse, you can indicate how you want to incorporate the object. At any time, you can choose to have COBWEB/3 use category utility instead of directing the incorporation yourself.

Incorporating an Object

The system classifies from the root of the hierarchy downward, and you direct the classification. Here is what COBWEB/3 does:

1. The object under current consideration displays in a separate object window.
2. The current level in the hierarchy is highlighted in the hierarchy window. The next incorporation will take place just below this level.
3. A separate static menu, the build-tree menu, displays beside the grapher menu. The build-tree menu has seven selections:

```
New Disjunct
Select Best
Merge
Split
CU-best
CU-best-whole-object
```

Here is an overview of what you should do:

1. Select an option on the build-tree menu with any mouse button.
2. Click on one or more nodes on the graph using the middle button.

Depending on your system's load, you might need to pause for a brief period between these two selections.

By selecting an option on the build-tree menu and then clicking on appropriate nodes in the graph, you incorporate the current object into the current level of the hierarchy. Here are the ways to incorporate, corresponding to the build-tree menu selections:

NEW DISJUNCT. Make the instance a new disjunct of the currently highlighted node in the hierarchy.

Nodes to select: none

SELECT BEST. Incorporate the instance into the node that you indicate.

Nodes to select:

1. BEST: middle mouse on any child of the highlighted node.

MERGE. Combine two nodes and incorporate the instance into the new merged node.

Nodes to select:

1. MERGE1: middle mouse on any child of the highlighted node.
2. MERGE2: middle mouse on another child of the highlighted node, sibling of MERGE1.

SPLIT. Replace a node with its children. This selection does not work exactly like the normal SPLIT operator; it simply splits a child of the currently highlighted node and allows the user to select what to do with the revised partition.[9]

Nodes to select:

1. SPLIT: middle mouse on child of highlighted node. The node you indicate must have children to be an appropriate selection.

CU-BEST. Incorporate the object into the node at the next level, as selected by category utility.

Nodes to select: none

CU-BEST-WHOLE-OBJECT. Incorporate the object into the remaining levels of the hierarchy, using category utility to guide the incorporation.

Nodes to select: none

Using the Hierarchy

Once you build the hierarchy, you can keep it for reuse in another COBWEB/3 run with the **Save This Tree** option on the grapher menu. For more information about this option, review Section 3.4.1.

9. As we interpret it, the original COBWEB Split operator simultaneously puts the object into the best of the split node's children.

References

Anderberg, M. (1973). *Cluster analysis for applications*. New York: Academic Press.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., & Freeman, D. (1988). AUTOCLASS: A Bayesian classification system. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 54–64). Ann Arbor, MI: Morgan Kaufmann.

Feigenbaum, E. A. (1963). The simulation of verbal learning behavior. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill.

Fisher, D. H. (1987a). Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, *2*, 139–172.

Fisher, D. (1987b). *Knowledge acquisition via incremental conceptual clustering*. Doctoral dissertation, Department of Information & Computer Science, University of California, Irvine.

Fisher, D. H., & Langley, P. (1986). Methods of conceptual clustering and their relation to numerical taxonomy. In W. Gale (Ed.), *Artificial intelligence and statistics*. Reading MA: Addison Wesley.

Fisher, D. H., & Langley, P. (in press). The structure and formation of natural categories. In G. H. Bower (Ed.), *The psychology of learning and motivation: Advances in research and theory* (Vol. 26). Cambridge, MA: Academic Press.

Gennari, J. H. (1989a). Focused concept formation. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 379–382). Ithaca, NY: Morgan Kaufmann.

Gennari, J. H. (1989b). *A survey of clustering methods* (Technical Report 89–38). Irvine: University of California, Department of Information & Computer Science.

Gennari, J. H. (1990). *Concept formation: An empirical study*. Doctoral dissertation, Department of Information & Computer Science, University of California, Irvine.

Gennari, J. H., Langley, P., & Fisher, D. H. (1989). Models of incremental concept formation. *Artificial Intelligence*, *40*, 11–61.

Gluck, M., & Corter, J. (1985). Information, uncertainty and the utility of categories. *Proceedings of the Seventh Annual Conference of the Cognitive Science Society* (pp. 283–287). Irvine, CA: Lawrence Erlbaum.

Iba, W., & Gennari, J. H. (in press). Learning movement concepts. In D. H. Fisher & M. Pazzani (Eds.) *Computational approaches to concept formation*. San Mateo, CA: Morgan Kaufmann.

Kolodner, J. L. (1983). Reconstructive memory: A computer model. *Cognitive Science*, *7*, 281–328.

Langley, P., Thompson, K., Iba, W. F., Gennari, J., & Allen, J. A. (1989). *An integrated cognitive architecture for autonomous agents* (Technical Report 89–28). Irvine: University of California, Department of Information & Computer Science.

Lebowitz, M. (1987). Experiments with incremental concept formation: UNIMEM. *Machine Learning*, *2*, 103–138.

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

Smith, E., & Medin, D. (1981). *Categories and concepts.* Cambridge, MA: Harvard University Press.

Michalski, R. S., & Stepp, R. (1983). Learning from observation: Conceptual clustering. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Thompson, K., & Langley, P. (1989). Incremental concept formation with composite objects. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 371–374). Ithaca, NY: Morgan Kaufmann.

Thompson, K., & Langley, P. (in press). Concept formation in structured domains. In D. H. Fisher & M. Pazzani (Eds.) *Computational approaches to concept formation.* San Mateo, CA: Morgan Kaufmann.

Yoo, J., Yang, H., & Fisher, D. H. (in press). Concept formation over explanations, plans, and of problem solving experience. In D. H. Fisher & M. Pazzani (Eds.) *Computational approaches to concept formation.* San Mateo, CA: Morgan Kaufmann.

Index

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dates attached	

4. TITLE AND SUBTITLE

Titles/Authors - Attached

5. FUNDING NUMBERS

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Code FIA - Artificial Intelligence Research Branch
Information Sciences Division

8. PERFORMING ORGANIZATION REPORT NUMBER

Attached

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Nasa/Ames Research Center

Moffett Field, CA. 94035-1000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Available for Public Distribution

Peter Friedland 5/14/92 BRANCH CHIEF

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Abstracts ATTACHED

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT