Chapter 5

Concept Formation in Structured Domains

Kevin Thompson
Pat Langley

1. Introduction

Most recent work on unsupervised concept learning has been limited to unstructured domains, in which instances are described by fixed sets of attribute-value pairs. Many domains can be described in this simple language. Frequently, however, instances have some natural *structure*; objects have components and relations among those components. In such domains, an attribute-value language is inadequate.

This chapter describes Labyrinth, an implemented system that induces concepts from structured objects. We view Labyrinth as an approach to *incremental concept formation*. Following Gennari, Langley, and Fisher (1989), we define this task as:

- *Given:* a sequential presentation of objects and their associated descriptions;
- *Find:* clusterings that group these objects into concepts;
- *Find:* a summary description for each concept;
- *Find:* a hierarchical organization for these concepts.

The goal of incremental concept formation is to find concepts that allow useful predictions from partial information. Cobweb (Fisher, 1987), Unimem (Lebowitz, 1987), and Cyrus (Kolodner, 1982) all incorporate approaches to this task, but these earlier systems are restricted to attribute-value languages. Labyrinth can make effective generalizations by using a more powerful structured representation language.

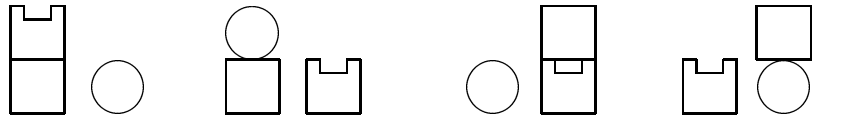

Figure 1. Four instances from a simple relational domain.

Figure 1 shows a simple domain with four structured objects. Each object has three components, two LEFT-OF relations, and a single ON-TOP relation. Each of the three component objects is in turn described with the two attributes SHAPE and COLOR. We will sometimes describe each object as being "labeled" as a member of one of these two classes: RIGHTSTACK (instances that have a two-high stack to the right of another object), and LEFTSTACK (instances with a stack to the left). These labels are for expository purposes; they play no part in classification or learning. We will use the domain in Figure 1 throughout our discussion of related work and our sketch of LABYRINTH's operation.

A structured domain can sometimes be converted to one described only by fixed attributes and their values. However, as Quinlan (1990) has argued, using an attribute-value language simplifies the learning task, but may prevent the learning of concise, effective generalizations. For example, if we enforce a consistent ordering of components in the four instances from Figure 1, one could in theory "flatten" the representation of each instance to a fixed set of attributes. However, to find the most concise representation of the two STACK concepts, the learner must be able to consider different bindings between the components of each object. With a structured representation, the learner can find a concept of the form: "There are three objects, X, Y, and Z; X is on top of Y, and both X and Y are to the left of Z". Moreover, a system that uses a structured representation has the potential to recognize a LEFTSTACK in which the stack has three objects instead of two; a learner that is limited to a predetermined set of attributes would have limited flexibility in such situations.

In the following section, we summarize related research on concept learning in structured domains. We then describe LABYRINTH's representation of objects and concepts, along with its memory organization. After this, we illustrate the system's classification and learn-

ing algorithm by tracing through a simple example, and then discuss Labyrinth's mechanisms for matching structured objects against concepts in memory. We conclude with a discussion of open issues and plans for future research.

2. Concept Learning in Structured Domains

Labyrinth carries out incremental, unsupervised concept learning in structured domains. It learns probabilistic concepts and uses them to make predictions of missing attribute values, components, and relations. It also decomposes objects into sets of components to constrain matching. Many of these characteristics are found in earlier systems, but no one system has integrated all these traits. In this section, we review six systems in detail. The review is not intended to be exhaustive, but to highlight previous work that has examined some of the issues Labyrinth addresses, and to argue for the importance of integrating these characteristics. Of the six systems we describe, two involve induction over objects described as attribute-value sets, and are important for their contribution to the unsupervised learning literature. The other four are important for their contribution to the understanding of induction in structural domains. For each system, we discuss its representation language for instances and concepts, its classification mechanism, and its learning algorithm.

2.1 Sprouter: Incremental Learning with Structured Objects

Hayes-Roth and McDermott's (1978) Sprouter is representative of several systems that carry out learning of maximally specific conjunctive descriptions from examples (e.g., Vere, 1975; Winston, 1975). These systems focus on finding characterizations, or descriptions, of classes given by an external teacher. Dietterich and Michalski (1981) present a careful comparison of a number of such systems.

Sprouter's representation language for both instances and concepts is equivalent to quantifier-free first-order predicate logic; the system views atoms as existentially quantified variables denoting distinct objects. In addition, the language allows organization of predicates that are semantically related into *case frames* to reduce match costs. For example, the Color feature of each object in a scene would be put

in the same frame.[1] SPROUTER's representation of the RIGHTSTACK-2 instance from Figure 1 would be:

```
{{grey:a,blue:b,red:c}
 {square:a,odd:b,circle:c}
 {on-top:a,below:c}
 {left:b,right:c}
 {left:b,right:a}}  .
```

This representation asserts that there is a structured object composed of three component objects, a, b, and c, that the object labeled a has properties of being GREY, SQUARE, and so on. Because SPROUTER induces only conjunctive generalizations, its concepts are represented in the same language as the instances. Generalization arises from dropping terms and replacing constants with variables.

SPROUTER uses an incremental algorithm to carry out a heuristic beam search[2] through the space of hypotheses using a specific-to-general scheme. It uses the first instance as an initial hypothesis set and creates new sets of conjunctive generalizations (e.g., concepts) in response to later instances. The system conducts an *interference match* between each generalization and each new instance. This match identifies common properties, and replaces the current hypothesis with one or more new generalizations. SPROUTER constrains its search through the hypothesis space by limiting the number of partial matches stored and pruning those with low utility; the evaluation function that guides this search is defined to increase with the number of relations in a match and to decrease with the number of objects related.

To match a structured instance I and concept C, SPROUTER selects an arbitrary case frame F_I from I and then finds bindings between F_I and a case relation F_C from C with identical case labels. As we noted, the system uses its case frames to guide its selection of F_C. SPROUTER uses F_I and F_C and the bindings between them to form an initial partial match; it then selects a new case frame from I and repeats the process. If the bindings between the frames are consistent with the previous

1. This case frame representation appears to play a role similar to that of the "attributes" used by many inductive learning systems. Hayes-Roth and McDermott (1978, p. 402) use this idea of defining certain shared properties. In addition, SPROUTER appears to use the case frames to direct matching between properties that other systems would represent as n-ary predicates (e.g., ABOVE, BELOW).
2. In contrast, Vere's THOTH (1975) considers *all* maximal generalizations.

partial match, SPROUTER adds the case frames and their bindings into the previous hypothesis. If the bindings conflict, the system forms a new generalization that contains only the bindings between the current case frames. In this way, each instance causes the system to extend or revise its set of hypotheses.[3]

SPROUTER is important because it is one of the most sophisticated of the early inductive systems able to learn in structured domains. It forms plausible characterizations in complex structured domains, using simple heuristic methods to limit an inherently exponential search problem for a maximal characterization. However, it is limited to a conjunctive concept language, and, like many early inductive learning systems, lacks a clear performance component. One can imagine SPROUTER being used in a recognition task in which the system uses a complete matcher to determine if a test instance matches any of the hypotheses. Most importantly for our current discussion, SPROUTER is a *supervised* algorithm, and therefore does not address issues of cluster formation and memory organization. We turn now to classification and learning in situations where the objects are unlabeled.

2.2 CLUSTER/2: Conceptual Clustering

In many situations, a learner cannot rely on direct labeling of each object; in these cases, one must autonomously organize observations into categories. Older work in this area, known as *numerical taxonomy* (Everitt, 1980), concentrates on what Fisher and Pazzani (Chapter 1, this volume) call the *clustering* task, that of determining useful subsets of an unclassified set of objects. With their system CLUSTER/2, Michalski and Stepp (1983) introduce the *conceptual clustering* paradigm. This task includes not only clustering, but also *characterization*: the formation of intensional concept descriptions from each extensionally defined cluster. This latter subtask is the focus of supervised learning systems such as SPROUTER; it is the combination of the clustering and characterization problems that distinguishes conceptual clustering.

3. Dietterich and Michalski (1981) divide the SPROUTER algorithm into two separate steps: finding all possible bindings between identical case frames, and finding consistent unions of them. Although this appears to be identical in principle to the description by Hayes-Roth and McDermott (1978), the latter argue (p. 405) that finding all possible bindings initially would be prohibitively expensive.

We review CLUSTER/2 here because it constitutes an early example of a machine learning approach to conceptual clustering, and is an important component of CLUSTER/S, which we describe in Section 2.3. However, we emphasize that the system does not carry out structured concept learning, since its representation language has only unary predicates. Both objects and concepts are represented in the *annotated predicate calculus*, an extension of the predicate calculus with additional operators for internal disjunction (e.g., $[shape(block) = odd \vee square]$) and internal conjunction (e.g., $[shape(block1 \& block2) = odd]$). Each predicate, variable, and function in this language has an associated *annotation*, giving "domain knowledge" about the type of the value set and related descriptors in a value hierarchy (e.g., ODD and SQUARE are subsumed by the value ANY-SHAPE).

CLUSTER/2 is nonincremental, using a divisive technique to generate a disjoint hierarchy of concepts. It starts with a root node consisting of all objects in the data set. It then splits that node into a set of mutually exclusive clusters and recurses to construct subhierarchies below each node. CLUSTER/2 is a complex algorithm, with several levels of nested search, each using a similar (user-supplied) evaluation function but using different search techniques. At the highest level, the algorithm searches through partitions consisting of different numbers of clusters k, from two up to a user-specified parameter K_{max}, finding a "best" partition for each value of k and then selecting the best of these $K_{max} - 1$ partitions.

The CLUSTER/2 system operates by transforming its unsupervised learning task into a series of supervised learning tasks. To find a partition for a single value of k, it begins by randomly selecting k "seed" objects for each seed, treating that seed as a positive instance and all others seeds as negative instances. For each seed, CLUSTER/2 uses the star-generating algorithm described by Michalski (1983) to find the set of alternative most general descriptions that distinguish the cluster based on that seed from those of the other seeds; it selects the best of these as the cluster for that seed.[4] These descriptions form a disjoint clustering over the original set of objects. If this iteration produces a set of clusters superior to the previous one, seeds are selected from the central tendency of each of these clusters; otherwise, seeds are selected from instances at the borders of the clusters. This new set of seeds is used to

4. Because they are the most *general* definitions, the clusters can overlap; an additional search is used to make them disjoint.

generate a new clustering, with the algorithm terminating when some predefined number of consecutive iterations generate no improvement.

CLUSTER/2 is interesting to our current discussion primarily because it introduces the task of conceptual clustering, and aids the discussion of CLUSTER/S below. The algorithm is computationally expensive and relies on several user-supplied thresholds to control its search. One would prefer the algorithm to determine the proper number of clusterings without a complete search for each value of k. In addition, like most early inductive learning systems, CLUSTER/2 lacks a performance mechanism with which to evaluate its clusterings, and instead relies on metrics like "quality of discovered classes" and "quality of fit to data" to evaluate the system's performance. However, as with SPROUTER, one can imagine using the induced concepts and a complete matcher to recognize test instances.

2.3 CLUSTER/S: Clustering with Structured Objects

CLUSTER/S (Stepp, 1984; Stepp & Michalski, 1986) extends CLUSTER/2, combining a supervised learning algorithm for structured domains with the earlier work on attribute-based conceptual clustering to form concept hierarchies from structured objects. CLUSTER/S represents objects and concepts in the annotated predicate calculus, as with CLUSTER/2, but includes n-ary predicates along with simple attributes. The RIGHTSTACK-2 object in Figure 1 would be represented as:

```
∃ p1,p2,p3 [color(p1)=grey] [shape(p1)=square]
           [color(p2)=blue] [shape(p2)=odd]
           [color(p3)=red]  [shape(p3)=circle]
           [on(p1,p3)]
           [Left-of(p2,p3)]
           [Left-of(p2,p1)]  .
```

Like its predecessor, CLUSTER/S effectively reduces an unsupervised learning problem to a series of supervised learning subproblems. The system breaks the problem of structured object clustering into two segments: reducing each object description to an attribute-value representation using a supervised learner, then using an attribute-based method to cluster these redescribed instances. It thus circumvents the complexity of clustering structural descriptions by clustering only those parts of each object expressible in a common language of fixed attributes.

CLUSTER/S first finds a maximally specific generalization, or *template*, of the set of structured objects, using a characterization algorithm adapted from INDUCE/2 (Hoff, Michalski, & Stepp, 1983). This generalization M expresses the common substructure of all the instances, covering all objects while preserving enough information from each object to identify correspondences between objects. Using M, CLUSTER/S can extract a subset of the literals from each instance in a common language of quantifier-free attributes. In this way, a structured domain is converted to an attribute-value language by a search for common structural properties. The re-defined objects, described by a fixed set of literals, are then clustered with the CLUSTER/2 algorithm, and these clusters can easily be converted back to a structured form using M. A postprocessing step augments each cluster with those parts of each instance "left out" in the conversion to the template language.

Stepp (1984) describes how the matching of two objects in structured domains can be viewed as a graph-matching problem, and notes its computational complexities. The algorithm used to generate M appears to employ a beam search through a space of partial matches for the instances, starting with a single attribute and gradually extending the set of template hypotheses by adding more attributes. The algorithm contains a "trimming" step to limit the combinatorial explosion of match hypotheses, but Stepp fails to describe clearly the evaluation function.

Many of the comments applicable to CLUSTER/2 are applicable to CLUSTER/S as well. The latter system is important as the first machine learning approach to unsupervised induction of concepts from structural data. However, because it uses CLUSTER/2 as a main subroutine, it shares the disadvantages of being computationally expensive and nonincremental. Like CLUSTER/2, it lacks a clear performance component. In addition, because it clusters only over those relations and attributes that are found in the template M, it cannot find generalizations that use features shared only by a subset of the instances.

2.4 Levinson's Incremental Self-Organizing Memory

Levinson (1985) describes a database retrieval system for concepts represented as graphs. He applies his system to the domain of organic chemistry, but argues that it is widely applicable, and demonstrates it briefly on chess. In contrast to the other systems we review, Levinson's system does not learn at the knowledge level (Dietterich, 1986), but aims to acquire efficient indices for retrieving specific cases.

Instances are represented as labeled graphs. For example, a hydrocarbon molecule would be represented as a graph with edges for bonds and vertices for individual atoms. Concepts are described as logical conjunctions of relations that share arguments, as in most work on structured concept learning. Naturally, concepts are partially ordered by generality, but Levinson's system uses this ordering for memory organization, not just to constrain search. The system stores all concepts in a graph partially ordered by the relation SUBGRAPH-OF. The most general nodes are individual literals; the most specific concepts (terminal nodes) represent actual objects. If one concept (S) is more specific than another (G), then S is connected to G by a SUBGRAPH-OF link, unless there is some other concept that is more general than S and more specific than G.

Levinson's system uses this memory organization to retrieve efficiently the best matches in memory to a presented object. A new instance I is sorted "in parallel" down all paths in the concept hierarchy, starting at the most general node. If I matches the concept C, then the instance is recursively sorted to C's children. This continues until I reaches a concept that it fails to match, or until it reaches a terminal node.

If an instance I reaches and matches a terminal node during sorting through memory, no learning occurs. However, if I has matched a concept G but does not match any of G's children, the system considers forming generalizations based on I and each of those children. For each child S, it finds all maximal partial matches between I and S, then selects the best match according to efficiency concerns. It creates a new intermediate level concept L that is more general than S and more specific than G, inserting the appropriate SUBGRAPH-OF links. The system also determines whether L should be inserted between any other pair of concepts that are directly connected by SUBGRAPH-OF links. Note that the system can create multiple concepts for a given instance I, since I is sorted down multiple paths in the hierarchy. However, the algorithm does *not* move beyond the input data, but only summarizes the observed instances. The algorithm forms general concepts, but only uses them as an efficient indexing scheme for retrieving specific cases. Wogulis and Langley (1989) use different mechanisms to acquire a similar memory structure, and point out that such systems lead to more efficient classification by storing intermediate concepts; in Levinson's system, the subgraphs allow more efficient indexing of structured objects.

2.5 MERGE: Organizing Structured Objects into Components

Wasserman (1985) describes MERGE, a system that carries out incremental concept acquisition and organization for structured objects. Like Levinson's system, it uses a memory organization to facilitate incremental update of memory in response to new objects. In contrast to Levinson's work, MERGE moves beyond the data, making generalizations that summarize instances and using those generalizations to fill in missing information.

MERGE's instance representation is most interesting to the current discussion. The standard representation for structured objects, a predicate calculus formalism, is equivalent to arbitrary directed graphs. Determining a match between two structured objects represented as graphs is equivalent to the NP-complete subgraph isomorphism problem. In response to this combinatorial problem, both SPROUTER and the CLUSTER programs use heuristics to control the search for a characterization. Wasserman takes an alternate approach: representing objects in a language in which generalizations are more easily found. MERGE is described as a system for learning from *hierarchies*, rather than from arbitrary structured objects. An instance hierarchy is represented as a tree of nodes partially ordered by a *fundamental relation*. This relation is used to decompose the structured object into smaller "components", which in turn can have components, and so on. The representation bottoms out with primitive objects described only by associated object properties. To distinguish these instances from concept hierarchies, we refer to instance hierarchies as *partonomies*.

In the domain of physical objects, the fundamental relation would be the PART-OF relation, but in other applications, Wasserman uses relations like REPORTS-TO (for human organization charts) and IS-A (for biological taxonomies). Wasserman notes that there are several possible organizational concepts, or fundamental relations, for any given domain, but argues that a single outstanding relation gives a complete partonomy of each object. This basic partonomy is augmented by *non-fundamental* relations, which are predicates other than the specified fundamental relation,[5] and which take as arguments any object in the instance tree. The RIGHTSTACK-2 object would thus be represented as:

5. It appears that MERGE is restricted to binary relations, although Wasserman never clearly states this constraint, and the extension to n-ary predicates seems straightforward.

```
(Rightstack-2 (component1 (color blue) (shape odd))
              (component2 (color red) (shape circular))
              (component3 (color grey) (shape square))
              ((Left-of component1 component3)))
              ((Left-of component1 component2))
              ((on component3 component2))  .
```

Wasserman uses a graphical notation for instances; we have substituted an equivalent syntax for purposes of comparison.

Generalizations are essentially the logical intersection of the instances from which they are made; the system avoids the extra search required to make disjunctive generalizations. Abstractions are made over the structural information (relations); Wasserman deemphasizes the importance of abstractions over object properties, although an unspecified algorithm does generalize the object properties.

MERGE incrementally forms abstraction hierarchies from a sequence of instance partonomies. The system classifies not only the entire structured object, but each of its subhierarchies[6] as well. Each of these subhierarchies is classified into a separate concept hierarchy, thus giving a forest of concept hierarchies, one for each "type" of object (apparently, each level of an instance is a different type). Like UNIMEM (Lebowitz, 1987), MERGE explicitly represents differences and similarities between a child and its parent with the use of inheritance to add, subtract, or substitute features. To classify each subhierarchy I of the instance, MERGE starts at the root of the concept hierarchy for that type of object and recurses through the tree. At each parent P, it finds the best candidate child C of that node. If C's score is no better than that of P, the algorithm stops and makes the object a new child of P. Otherwise, it incorporates I into C and recurses. MERGE's evaluation function is a scoring scheme relying on several heuristics. Two components that have a common ancestor in a concept hierarchy are rewarded if that ancestor is low in the partonomy, and components are scored based on their literal similarity. In addition, components are weighted less in the overall score than the object itself.

Wasserman (1985) downplays the computational difficulties of matching structured objects. The augmented partonomy representation of MERGE lends itself to decomposition of the match problem into a series

6. Remember that these are PART-OF hierarchies that represent individual instances, not concept hierarchies.

of component-matching problems, unlike the arbitrary graph representation used by earlier systems. However, Wasserman does not promote this as an advantage of using partonomies. Although the system classifies each subtree of the instance, it does not use the results of component classifications in classification of instances. It thus faces the problem of generating abstractions from arbitrary trees. The MERGE matcher compares two trees by working its way bottom up through each partonomy, finding "best" matches at each level and recursing. It appears to use an exhaustive matcher that matches m instance components against n concept components, with a computational complexity of $O(n!)$. In addition, the matcher has operators for "level hopping" that involve checking whether a component at level x of the one partonomy matches well against a component at a different level y of another partonomy.

2.6 COBWEB: Probabilistic Concepts

Because the COBWEB system (Fisher, 1987; McKusick & Thompson, 1990) forms the basis for LABYRINTH, we review it in some detail. COBWEB is an incremental, unsupervised concept learner, like CYRUS (Kolodner, 1983) and UNIMEM (Lebowitz, 1987). It differs from its predecessors in its use of *probabilistic* concepts (Smith & Medin, 1981) and its use of a principled evaluation function that favors clusters that maximize the potential for inferring information. In addition, Fisher emphasizes the use of concept formation systems in the context of a performance task — missing attribute prediction — and explicitly evaluates his system using this task. This contrasts with most earlier unsupervised learners, which have been evaluated only in light of the concepts formed and their "comprehensibility".

COBWEB represents each instance as a set of nominal[7] attribute-value pairs, and it summarizes these instances in a hierarchy of probabilistic concepts. Each concept C_k is described as a set of attributes A_i and their possible values V_{ij}, along with the conditional probability $P(A_i = V_{ij}|C_k)$ that a value will occur in an instance of a concept. The system also stores the overall probability of each concept, $P(C_k)$. Thus, whereas CLUSTER/2 can represent an attribute *color* with alternate values $blue \vee red$, a COBWEB concept can represent the observed conditional

7. Gennari, Langley, and Fisher (1989) describe CLASSIT, a variant of COBWEB that accepts real-valued attributes. LABYRINTH's mechanisms are independent of the feature types of primitive object attributes.

probabilities, $P(color = blue|C_k) = 0.6$ and $P(color = red|C_k) = 0.4$. The use of probabilistic concepts is crucial to COBWEB's design. As Hanson and Bauer (1989) point out, many categories are better represented as probabilistic concepts than as sets of common features. For incremental systems with a restricted hypothesis memory, probabilistic concepts are crucial to avoid brittleness in the face of noisy or approximate concepts. Probabilistic concepts allow gradual updating of descriptions and recovery from misleading training orders because they store more information about the instances that form the concept.

COBWEB organizes its acquired concepts in a probabilistic concept hierarchy, in which each node is indexed by IS-A links from its parents, rather than difference links as with UNIMEM and MERGE. Specific instances are stored as leaves of the concept hierarchy, and the root node summarizes all instances seen in the domain. Such hierarchies are crucial for focusing attention and allowing small local changes to memory during incremental processing.

The system integrates classification and learning, sorting each instance through its concept hierarchy and simultaneously updating memory. Upon encountering a new instance I, COBWEB incorporates it into the root of the existing hierarchy and then recursively compares the instance with each new partition as I descends the tree. At a node N, the system considers incorporating the instance into each child of N as well as creating a new singleton class, and evaluates each resulting partition. If the evaluation function prefers adding the instance to an existing concept, COBWEB modifies the concept's probability and the conditional probabilities for its attribute values and then recurses to the children of that concept. If the system decides to place the instance into a new class, it creates a new child of the current parent node, and the classification process halts. COBWEB also incorporates two bidirectional operators, splitting and merging, that make local modifications to the hierarchy structure. These mitigate sensitivities to instance orderings, giving the effect of backtracking in the space of concept hierarchies without the memory overhead required by storing previous hypotheses.

To choose among these operators, COBWEB uses the probabilistic information stored in memory in an evaluation function — *category utility* — which favors high intra-class similarity and high inter-class differences. Gluck and Corter (1985) derive this function from information theory, and Fisher modifies it slightly to control classification and learn-

ing behavior in COBWEB. Given a set of n categories, category utility is defined as the *increase* in the expected number of attribute values that can be correctly guessed over the expected number of correct guesses without such knowledge. The version used by COBWEB is

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

where k varies over categories, i over attributes, and j over values for each attribute. This function evaluates a *partition* — defined as a parent node C and its immediate children C_k. The term $P(C_k)$ refers to the *a priori* likelihood that an instance is a member of the child C_k, whereas $P(A_i = V_{ij}|C_k)^2$ is a measure of *within-class similarity*, that is, how well the instances summarized by C_k resemble one another. The subtraction of the parent's within-class similarity $P(A_i = V_{ij}|C)^2$ lets category utility measure the information gained by partitioning the parent class into a set of children. Dividing by K, the number of C's children, biases the system against proliferation of singleton classes.

COBWEB has many positive characteristics, many of which will be important to the design of LABYRINTH. Its well-defined performance task, tightly integrated with its learning component, allows evaluation of the concepts learned. Its use of probabilistic concepts and a single evaluation function allows more robust performance than earlier concept formation systems. Its simple local reorganization operators give it the partial ability to overcome misleading orders of training instance with minimal reprocessing of previous instances. However, COBWEB can only learn in domains in which there are a finite number of unstructured attributes; LABYRINTH builds on COBWEB to overcome this limitation.

2.7 Issues in Structural Learning

Table 1 summarizes the six systems we have just reviewed, as well as LABYRINTH, across five important characteristics. LABYRINTH is the only system that exhibits all five characteristics: it is an incremental, unsupervised learning method that acquires probabilistic concepts from relational data, using the heuristic of breaking the instance into components for classification. From our review, we can see the origins of these ideas. SPROUTER and related systems were the earliest to face the problem of learning in structured domains. These programs are supervised,

Table 1. Issues addressed by LABYRINTH and its predecessors.

System	Probabilistic	Incremental	Unsupervised	Relations	Components
SPROUTER		⊕		⊕	
CLUSTER/2			⊕		
CLUSTER/S			⊕	⊕	
Levinson		⊕	⊕	⊕	
MERGE		⊕	⊕	⊕	⊕
COBWEB	⊕	⊕	⊕		
LABYRINTH	⊕	⊕	⊕	⊕	⊕

and learn only a single conjunctive concept at a time, avoiding issues of memory organization but explicitly proposing algorithms to form generalizations from multiple objects described in a structured language.

We have seen that CLUSTER/2 differs from most earlier inductive learning algorithms in that it is *unsupervised*; it discovers object classes and characterizes these classes as well. Its successor, CLUSTER/S, incorporates the advances of CLUSTER/2, but uses a structured object and concept language. Unfortunately, both these systems are nonincremental, requiring all instances in order to generate classes. Levinson is among the first[8] to propose a method for incrementally generating a memory organization containing structured concepts from unclassified instances. However, his system, in using what is in effect a complete matcher, fails to go beyond the data and to enlarge the deductive closure of its knowledge base. Wasserman's MERGE can be viewed abstractly as a version of Levinson's system that uses a partial matcher, and thus makes accurate classifications of previously unseen instances. In addition, Wasserman introduces the heuristic of decomposing structured objects into a tree, thus using one fundamental relation to organize memory and direct learning.

The basic classification mechanism and memory structure of COBWEB anticipates that of the current work. This system's use of probabilistic concepts gives it power to make more effective predictions than earlier

8. EPAM (Feigenbaum, 1963) also acquires concepts from hierarchically decomposed objects, but this system does not handle relations among components.

systems. In addition, COBWEB adopts prediction as a performance task for unsupervised learning systems. We have noted that COBWEB has many good characteristics, but is limited to attribute-value languages.

3. Representation and Organization in LABYRINTH

Having described earlier systems that address many of the issues faced by LABYRINTH, we are ready to discuss the current system at length. We will see that LABYRINTH has distinct ties to COBWEB, adopting its basic principle of probabilistic concepts organized in a disjoint hierarchy, and its divisive concept formation algorithm. However, the current system extends the representation language for objects and concepts. As we have seen, a central obstacle to learning in structured domains is that of characterizing structured concepts. LABYRINTH uses a representation for structured objects that reduces search by decomposing structured objects into a *partonomy* of components,[9] supplemented by additional relations among those components. In this section, we describe the system's representation for objects and concepts, and how these concepts are organized in long-term memory.

3.1 Instances in LABYRINTH

Following Wasserman (1985), we argue that in many domains the instances passed to a concept learner are naturally decomposed by a *fundamental relation*. For example, Marr (1982) has argued that the visual system parses physical object descriptions into a partonomy organized by PART-OF relations. Similarly, McNamara, Hardy, and Hirtle (1989) have found that memory for large-scale spatial environments has a hierarchical component. Many forms of sequential data also can be represented as an ordered set of components; Rubin and Richards' (1985) work on elementary motion boundaries presents evidence that humans perceive motion in distinct segments that are invariant with respect to speed and viewpoint. For continuity, we use physical objects for our example instances, but we will discuss alternative domains and fundamental relations in Section 5.3.

9. Recall from Section 2.5 that we use the term *partonomy* for object hierarchies, to distinguish them from concept hierarchies (taxonomies).

LABYRINTH treats one relation as fundamental and structures both objects and concepts by that relation. A structured object is represented as a partonomy whose constituents are linked together by the fundamental relation. Each object can be augmented by non-fundamental relations whose arguments are components of that object. Consider again the domain shown in Figure 1. We represent the rightmost instance in Figure 1 as:

```
(Rightstack-2 (component1 (color blue) (shape odd))
              (component2 (color red) (shape circular))
              (component3 (color grey) (shape square))
              ((Left-of component1 component3)))
              ((Left-of component1 component2))
              ((on component3 component2))  .
```

Note that the PART-OF relation is implicit in this representation and is used to organize the object into a partonomy, as in MERGE.

We distinguish between two types of objects. *Primitive* objects are leaves of an instance partonomy. They are represented as ordered sets of attributes whose values are directly observable object features, as in COBWEB. For example, RIGHTSTACK-2 has three primitive components: COMPONENT$_1$, COMPONENT$_2$, and COMPONENT$_3$. *Structured* objects are represented as unordered sets of attributes (components) whose values can additionally be either primitive objects or other structured objects. Here, RIGHTSTACK-2 is a structured object with three attributes, each of which has a value that is a primitive object. In addition, this object has three associated binary relations, ON and two different instances of LEFT-OF, which are treated as additional attributes during classification. LABYRINTH treats components, non-fundamental relations, and descriptive features as different forms of "attributes". It exploits the isomorphism among them to classify both primitive and structured objects using a similar algorithm.

3.2 Concept Representation and Organization in LABYRINTH

Like COBWEB, LABYRINTH represents concepts by storing an associated set of attributes, their values, and associated conditional probabilities; it differs by the types of data that can be tied to those attributes. We define a *primitive* concept as a concept whose attributes have directly observable values. In contrast, a *structured* concept is one whose

attributes correspond to "components".[10] Because these components are themselves objects, a structured concept's "attributes" have associated values that point at other concepts summarizing those objects. In this way, a single structured concept is defined in terms of other, possibly structured, concepts. A structured concept is thus stored not as a monolithic structure, but as many concepts distributed through memory, decomposed by the fundamental relation for that domain.

Because of this distributed representation, a single component concept can take part in several structured concepts. The concepts "pointed to" are themselves acquired by LABYRINTH, so that one can view the system as learning new terms; only primitive concepts are represented with values present in the original instance language. In addition, because all the concepts are changing over time in response to new information, LABYRINTH can manage concept drift with respect to both structured object concepts and component concepts.

Figure 2 shows a snapshot of LABYRINTH's memory after it has incorporated three instances into memory: LEFTSTACK-1, LEFTSTACK-2, and RIGHTSTACK-1. The two singleton children of the LEFTSTACK concept, as well as the mixed root, are omitted for brevity. Each concept has been given a name for expository purposes, and each has an associated probability $P(N)$ with respect to its parent, along with a set of attributes. Each of these attributes in turn has a set of values and associated conditional probabilities. Note that for some of the concepts these associated values are in italics to indicate that they are the names of other concepts in memory. Thus, the hierarchy of Figure 2 contains thirteen primitive concepts, which represent stack components, and five structured concepts (of which three are shown), which represent stacks. Concepts for both are indexed in the same memory structure. The root concept thus summarizes both stacks and stack components, and is used only as an index for the hierarchy.

In addition to components, structured concepts can have arbitrary relations associated with them. LABYRINTH represents these relations as ternary-valued attributes, with associated conditional probabilities for each of the possible situations CONFIRMED, NEGATED, and MISSING. If a relation is not found in an object description, the system

10. Some concepts are "mixed", in that they generalize both primitive and structured objects; for example, the root of the tree will always be mixed. Because primitive and structured objects never have values in common, these mixed concepts rarely appear below the first level of the tree in LABYRINTH runs.

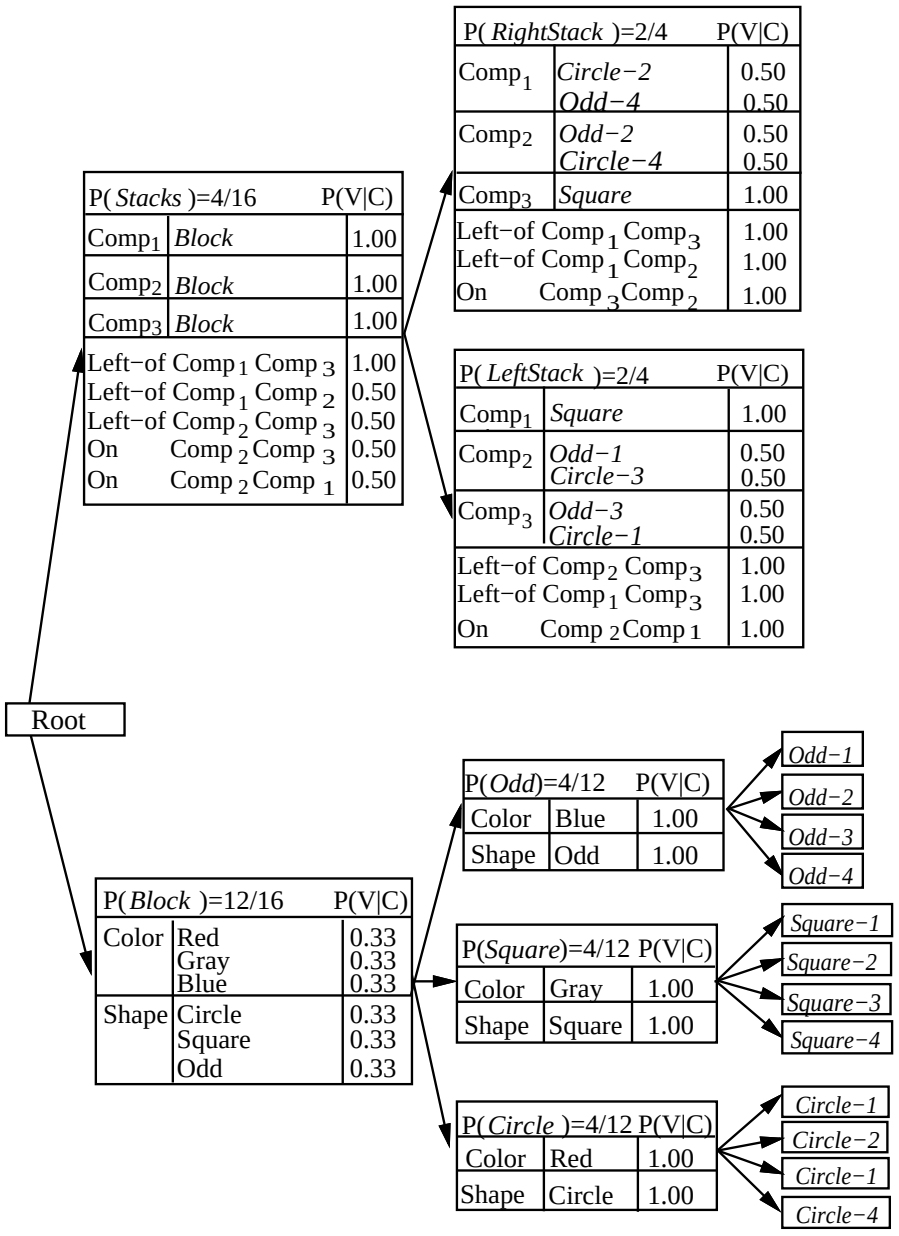

Figure 2. LABYRINTH's memory after processing three instances from Figure 1.

increments its MISSING probability. For brevity, we include only the CONFIRMED probability in our figures, but all three values are used for classification decisions. Note that there can be several different instantiations of the same relation at a concept; both (LEFT-OF COMPONENT$_1$ COMPONENT$_3$) and (LEFT-OF COMPONENT$_1$ COMPONENT$_2$) are associated with the RIGHTSTACK concept.

4. Classification and Learning in LABYRINTH

Having described LABYRINTH's memory structures, we can now describe how it classifies objects and updates its concept hierarchy. As in COBWEB, classification and learning are intertwined, with each instance being sorted through a concept hierarchy and altering that hierarchy in its passage. The system initializes its hierarchy to a single node based on the first instance. It then enters a loop of accepting new instances, classifying them and updating memory along the classification path. LABYRINTH differs from COBWEB in two important ways. It adds an outer loop to classify each component of a structured object. In addition, it introduces a new subroutine, COBWEB$'$, to form predictive characterizations of structured concepts.

4.1 The LABYRINTH Algorithm

Table 2 shows the top-level LABYRINTH algorithm for classifying and learning with structured objects. To classify a single instance, the system uses a divide-and-conquer technique, breaking up the overall classification problem into a series of simpler classifications, one for each subtree of the instance partonomy. LABYRINTH processes structured objects in a "component-first" style, performing a complete postorder traversal of the partonomy. To classify a structured object in the partonomy, the system first classifies each of the object's components, returning the node in memory that the component most closely matches. LABYRINTH then "re-labels" the structured object, using each returned node as a label for a component. By performing this re-labeling operation, LABYRINTH reduces a structured object to a simple one with attributes and corresponding values; however, the values in this case are pointers to nodes in memory. The system then classifies this re-labeled object and recurses, until all the structured objects of the instance, including the instance itself, are classified.

Table 2. The basic LABYRINTH algorithm.

```
Input: OBJECT is a composite object, with substructure given.
       ROOT   is the root node of the concept (is-a) hierarchy.
Side effects: Labels OBJECT and all its components with class names.

Procedure Labyrinth(OBJECT, ROOT)
   For each primitive component PRIM of composite object OBJECT,
      Let CONCEPT be Cobweb(PRIM, ROOT);
      Labyrinth'(OBJECT, PRIM, CONCEPT, ROOT).

Procedure Labyrinth'(OBJECT, COMPONENT, CONCEPT, ROOT)
   Label object COMPONENT as an instance of category CONCEPT.
   If COMPONENT is not the top-level object OBJECT,
      Then let COMPOSITE be the object that contains COMPONENT.
         If all components of COMPOSITE are labeled,
            Then let COMPOSITE-CONCEPT be Cobweb'(COMPOSITE, ROOT).
              Labyrinth'(OBJECT, COMPOSITE, COMPOSITE-CONCEPT, ROOT).
```

LABYRINTH uses two principal subroutines. The first of these is Fisher's COBWEB, which we have described in Section 2.6. LABYRINTH uses COBWEB to classify primitive components, treating it as a black box that returns the best match in memory to the object passed to it; we refer to this match as the *label* for that component. LABYRINTH relies on a second subroutine, COBWEB′, to classify non-primitive objects. This routine is based on COBWEB; it uses the same evaluation function, basic control structure, and learning operators. However, COBWEB′ incorporates additional mechanisms for finding the characterization of structured concepts.

We first illustrate LABYRINTH's processing on a simple two-level instance, RIGHTSTACK-2, from Figure 1. We then describe COBWEB′ and its mechanisms for characterizing structured concepts.

4.2 LABYRINTH Classifying a Structured Object

We start with memory as in Figure 2, after three instances (two of LEFTSTACK and RIGHTSTACK-1) have been processed. To process the new instance RIGHTSTACK-2, LABYRINTH passes the description of COMPONENT$_1$ to COBWEB, which classifies and returns a label (the concept

ODD-4) for that component. The same procedure leads LABYRINTH to label COMPONENT_2 as a member of CIRCLE-4, and COMPONENT_3 as a member of SQUARE-4. So far, LABYRINTH has done no more than use COBWEB's existing mechanisms to "label" three primitive objects and update memory accordingly. However, whereas COBWEB stops there, LABYRINTH *uses* this information to classify the structured object. These labels are inserted into the structured object description, so that the instance now has the form:

```
(Rightstack-2 (component1 odd-4)
              (component2 circle-4)
              (component3 square-4)
              ((Left-of component1 component3)))
              ((Left-of component1 component2))
              ((on component3 component2))  .
```

LABYRINTH treats these labels from previous classifications as nominal values, enabling it to classify the structured object as though it were a primitive object (with the exceptions described in Section 4.3). In this case, LABYRINTH labels the structured object as a member of the structured concept RIGHTSTACK, resulting in the memory structure found in Figure 3. Here, we see that the concepts labeled BLOCKS, SQUARE, ODD, and CIRCLE have been updated in response to the components of RIGHTSTACK-2, and new leaves have been added to the concept tree for ODD-4, CIRCLE-4, and SQUARE-4. In addition, the stack itself has passed through the STACKS and RIGHTSTACK concepts, updating them accordingly. As in Figure 2, we omit the singleton concepts for the individual stacks, indexed by the LEFTSTACK and RIGHTSTACK concepts.

4.3 Integrating a Structured Object into a Concept

As we have seen in Section 2, the primary difficulty in learning from structured data is finding adequate *characterizations* of the concepts. LABYRINTH has a simplified characterization task because it learns from trees, not from the arbitrary graphs used by programs like SPROUTER and CLUSTER/S. However, LABYRINTH's subroutine COBWEB′ still faces two extra searches to form characterizations. First, as we have noted, in many structural domains the components are *unordered*; in addition, whereas each object in Figure 1 has an identical number of components, some domains have objects with varying numbers of components. A

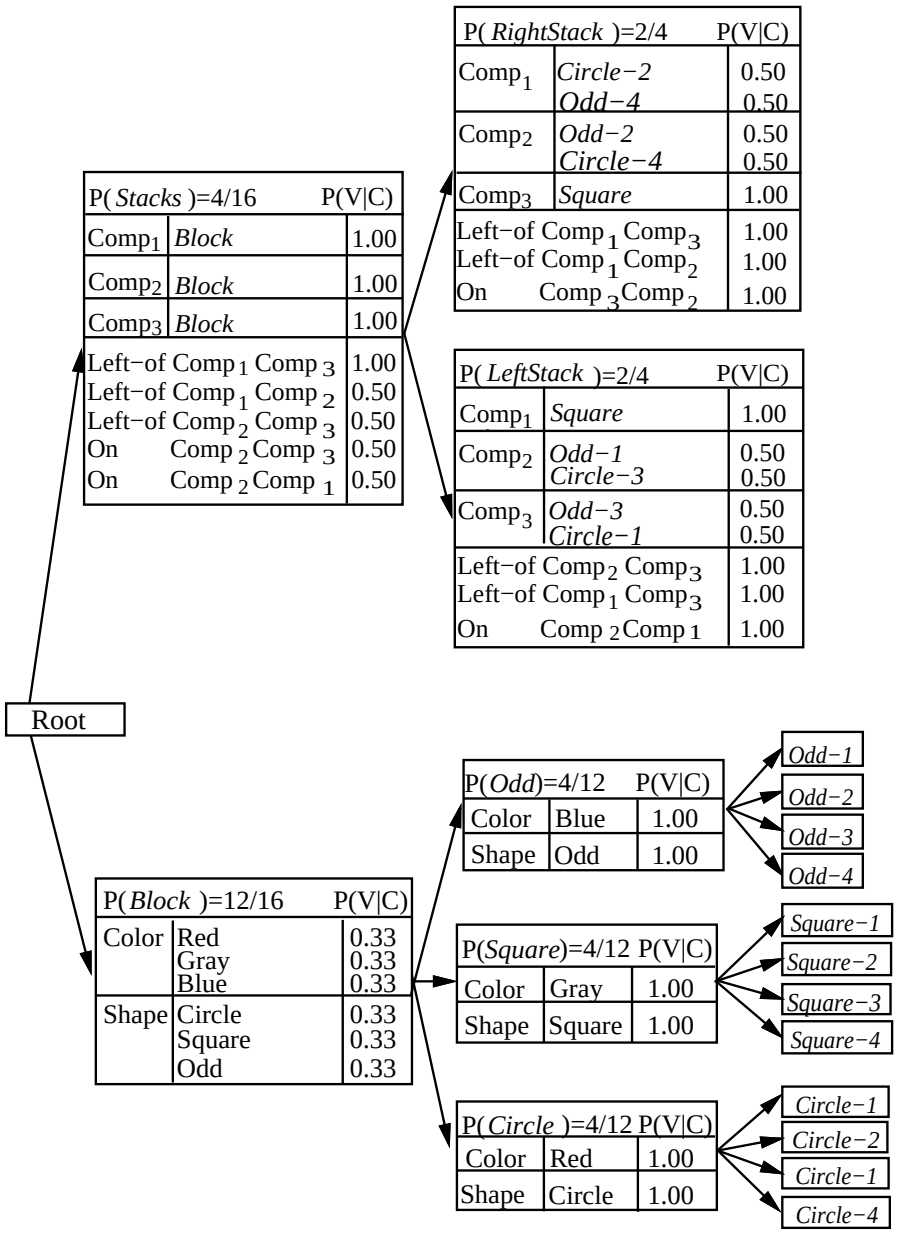

Figure 3. LABYRINTH's memory after processing four instances from Figure 1.

Table 3. Incorporating a structured object into a structured concept.

```
Variables: NODE is a node in the hierarchy.
           INST is an unclassified structured object.

Incorporate(NODE, INST)

   Update the probability of category NODE.
   Let BINDINGS be all possible bindings between NODE and INST.
   For each possible set of bindings BIND in BINDINGS,
      For each relation REL in instance INST,
       Bind the arguments of REL according to BIND.
       If there is an equivalent relation N-REL in NODE,
            Then update the probability of N-REL;
            Else add REL to the characterization of NODE.
      For each attribute ATT in instance INST,
         Let I-VAL be the value of ATT.
         Let N-ATT be the corresponding attribute in NODE.
         For each value VAL of N-ATT,
            Update the probability of VAL given category NODE.
         Let VALS be the value list of N-ATT.
         Let the VALS be Attribute-Generalize (I-VAL, N-VALS, NIL).
      Evaluate the resulting node.
   Choose the best possible BIND and use it to store INST.

Attribute-Generalize(OBJ-VAL, NODE-VALS, CHECKED)

   For each VALUE in NODE-VALS,
      Let ANCESTOR be the common ancestor of OBJ-VAL and VALUE.
      Let REST be NODE-VALS with VALUE removed.
      If it is appropriate to replace OBJ-VAL and VALUE with ANCESTOR,
         Then call Attribute-Generalize(ANCESTOR, REST, CHECKED);
         Else call Attribute-Generalize(OBJ-VAL, REST, CHECKED).
```

characterization algorithm for structured concepts must thus determine a set of bindings between components in the object and those in the concept.

The second search arises from the nature of objects that COBWEB′ processes. The routine classifies objects that are "re-labeled", in that their values are labels returned by earlier classifications. COBWEB′ takes advantage of the hierarchical relationships between these labels to search for more predictive characterizations of structured objects. In this sec-

tion, we describe the approach COBWEB′ takes to searching for the best characterization. Table 3 summarizes the function for incorporating an object into a concept in memory. This function replaces the simpler one used in COBWEB (see Gennari et al., 1989).

4.3.1 MATCHING COMPONENTS AND BINDING RELATIONS

Structured objects have unordered attributes; they lack *role* information that helps the learner match components from different objects. Whereas for primitive objects there is a unique maximal generalization for any pair of objects, structured objects require an extra matching process in order to determine the best match between the components and relations in the object and those in the concept. COBWEB′ must bind the arguments of object relations to determine if they match with one of the concept relations. Once it has bound the arguments of each object relation, COBWEB′ can update the correct probability (either CONFIRMED, NEGATED, or MISSING) of each concept relation based on the bound object. These concept relations are then treated as additional attributes of the concept in classification decisions.

We describe here an exhaustive algorithm to find the best mapping between a structured object and a structured concept.[11] COBWEB′ matches a structured object with each concept in memory using a four-step process. First, it finds all mappings of components in the object to components in the concept. Second, for each mapping, the system rewrites the n-ary relations in the object by substituting each object component for its corresponding concept component. Third, COBWEB′ compares the resulting instantiated relations to the relations in the concept description, treating each one as a Boolean attribute that may or may not match the concept. Finally, the system considers applying the attribute generalization operator described in Section 4.3.2 to each attribute. The resulting concept, with fully bound relations and attributes, is evaluated with a reduced form of category utility that evaluates the quality of a single concept:

$$\sum_{i}^{Atts} \sum_{j}^{Values} P(A_i = V_{ij}|C_k)^2 \quad . \tag{2}$$

11. For n components, this algorithm is $O(n!)$. Clearly, such a solution is impractical, and fails to take advantage of the simpler matching problem faced by COBWEB′. We discuss some less expensive solutions in Section 5.1.

This expression rewards matches that reinforce values already found in a node. The system selects the mapping that produces the node with the best score.

For example, when COBWEB′ matches the fourth instance against the RIGHTSTACK concept from Figure 2, there are $3! = 6$ mappings between the three components of the instance and the three components of the concept. Some of these mappings reinforce the values in the components but not the associated relations in the concept; others reinforce the relations but not the components. LABYRINTH includes both components and relations as attributes in Equation 2. For the RIGHTSTACK concept, the system chooses an instantiation that reinforces all three relations found in the instance. This generates bindings for both the components and relation arguments in the object, producing the characterization found in the RIGHTSTACK concept of Figure 3.

4.3.2 ATTRIBUTE GENERALIZATION IN LABYRINTH

COBWEB′ uses an additional mechanism to determine appropriate abstractions. For COBWEB, in which attributes take on only symbolic values, updating an attribute A_i after inspecting a new object is a simple matter of updating the correct conditional probability $P(A_i = V_{ij})$. However, in COBWEB′, the values in the object are concepts stored elsewhere in the hierarchy (the results of previous classifications). In order to determine the best generalization between this structured object and an existing structured concept, COBWEB′ uses the hierarchical relationships between the values in the object and those in the existing concept to determine the best values for the updated concept.

When incorporating an object into a concept, COBWEB′ first adds the label from the object to the corresponding attribute A_i (as found by a step of the match process) in the concept, resulting in a set of values V_i. The system then evaluates whether to apply *attribute generalization* to the values on each attribute. We define attribute generalization as replacing a subset of the values V_i stored at attribute A_i with their common ancestor, resulting in a smaller set of values. Attribute generalization chooses between two possibilities. In the simple case, the operator can leave V_i intact, as would COBWEB. The COBWEB′ routine also considers replacing a subset W_i of V_i with its common ancestor W_i^*. This results in a structured concept that can match more objects

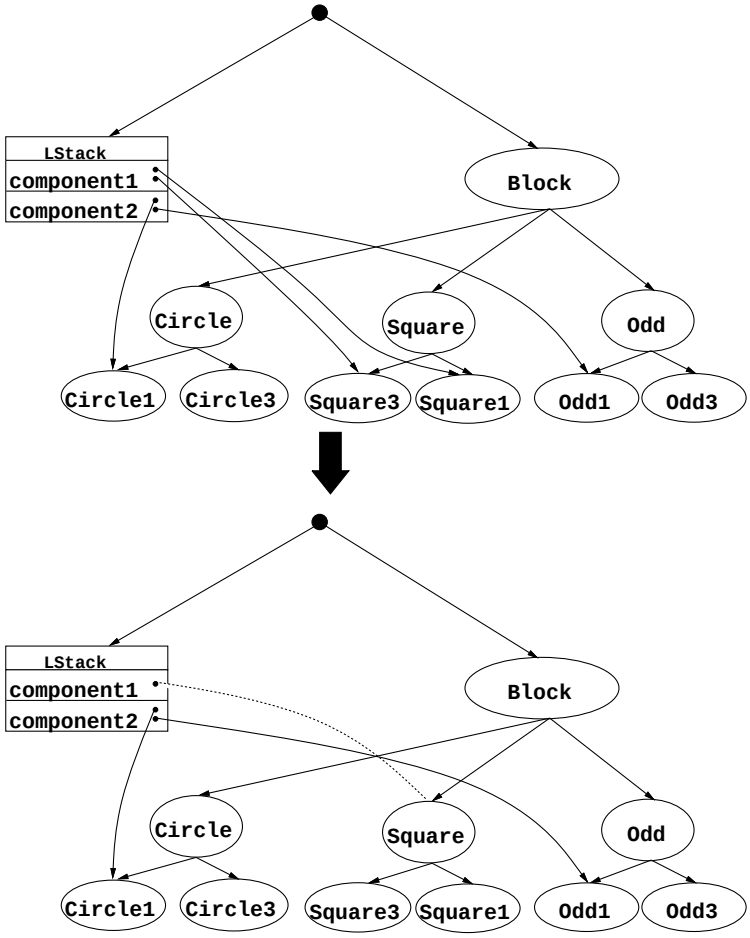

Figure 4. Generalizing the values of an attribute.

because of its more general values. For example, in Figure 3, we can see that for the attribute COMPONENT$_2$ in the concept RIGHTSTACK, COBWEB$'$ has simply added the object label CIRCLE-4 to the previous label ODD-2. In contrast, note that COMPONENT$_3$ has the single value SQUARE, the common ancestor of the two values SQUARE-4 (from RIGHTSTACK-2) and SQUARE-2 (from RIGHTSTACK-1). This reflects the fact that the two objects of RIGHTSTACK that LABYRINTH has classified to this point have a square stacked on another item. Figure 4 illustrates the application of this operator.

Determining when to apply the attribute generalization operator requires evaluating a tradeoff. Equation 2 favors a single value with high probability over several values with lower probability. Its application to attribute generalization would result in storing each attribute with a value V_i^* (the common ancestor of all the disjuncts) and probability of 1. While this would result in a higher score for Equation 2, it would result in less predictive power for the partition, because it would be difficult to discriminate such a node from other overgeneralized concepts. COBWEB′ evaluates the tradeoff between forming concise characterizations with few values at one extreme, and over generalizing values so that concepts cannot be distinguished at the other extreme. It replaces a value set V_i with a shorter set W_i if doing so increases the information gain between a child C_k and its parent C. For an attribute A_i in C_k, COBWEB′ replaces a set of values V_i with a new set of values W_i iff:

$$\sum_{l \in W_i} [P(A_i = W_{il}|C_k)^2 - P(A_i = W_{il}|C)^2] \geq \sum_{j \in V_i} [P(A_i = V_{ij}|C_k)^2 - P(A_i = V_{ij}|C)^2] \; . \quad (3)$$

The left side of this equation measures the information gain if the values are generalized at both the child C_k and the parent C. The right side measures the gain if the values are left as an internal disjunct.[12] COBWEB′ considers all new value sets W_i that are strictly more general than the original set V_i, and stores with C_k the first set it finds for which Equation 3 holds.

In one sense, this attribute generalization process is simply an incremental approach to learning with *structured attributes* through climbing a "generalization tree", as described by Michalski (1983), Mitchell, Utgoff, and Banerji (1983), and others. However, recall that LABYRINTH is constantly revising the structure of its concept hierarchy and introducing new symbols as it acquires new concepts. Since the descriptions of structured concepts refer to other concepts in the concept hierarchy, which LABYRINTH has acquired, the attribute generalization process operates over different knowledge structures at different points in the learning process. In effect, LABYRINTH is *dynamically changing* the representation used to describe its structured concepts.

12. Doug Fisher (personal communication) suggested the use of Equation 3.

5. Discussion

We believe that LABYRINTH constitutes a promising approach to concept learning in structured domains. However, the existing system has a number of limitations that we plan to remedy in future efforts. In addition, we must demonstrate the system on a variety of domains. We discuss these issues below, along with some related work.

5.1 LABYRINTH, Partial Matching, and Analogy

We have described an exhaustive algorithm to match components of an object to those of a concept in memory. Although this $O(n!)$ algorithm is guaranteed to find the optimal match according to Equation 2, more efficient alternatives exist. We plan to examine the Hungarian algorithm (Papademetriou & Steiglitz, 1982), a guaranteed matching algorithm that uses additional space to save partial solutions. Given a bipartite graph with $2n$ concepts, along with some function for evaluating the quality of a match, the Hungarian method finds the best match in $O(n^3)$ time. The algorithm works by creating an $n \times n$ cost matrix for all possible pairs of components and then solving an "n rooks" problem over this matrix. We are also studying the use of a heuristic beam search (e.g., as used by SPROUTER) guided by Equation 2. We plan to use background knowledge about the data types of components to constrain this search as well.

LABYRINTH can be viewed as an approach to *partial matching* (Hayes-Roth, 1978). This task is usually defined as a comparison of two descriptions to identify their similarities, and is thus typically used in systems that use a specific-to-general search for hypotheses. SPROUTER (Hayes-Roth & McDermott, 1978) and its relatives (Winston, 1975; Vere, 1975) all use partial matchers as a principal subroutine in their search for generalizations. Some additional work has focused on partial matching outside the context of a concept learner. Kline (1981) emphasizes ordering the space of possible partial matches to reduce computation. In contrast, Watanabe and Rendell (1990) reduce computation by finding branches of the search tree that can be eliminated without loss of information by pruning redundant paths.

In determining the best match between a structured object and a structured concept, LABYRINTH is performing a crucial subtask in analogical reasoning. Falkenhainer, Forbus, and Gentner (1989) have de-

veloped the Structure-Mapping Engine (SME) for determining this best match, emphasizing structural integrity of the structured object over object similarity. Matching is constrained in structure mapping by higher-order relationships that are included as part of the instance. In contrast, LABYRINTH treats relations and object attributes as two contributors to the same evaluation function, rather than treating relations as preeminent. This should allow it to find some generalizations based on surface features that SME would not consider. However, the current version of LABYRINTH uses a far less constrained matcher than SME and will thus require far more computation in complex situations.

5.2 Using Context in Classifying Components

The current version of LABYRINTH takes a purely "component-first" or "context-free" approach to structured classification. An alternative approach would be to take context as the primary criterion, classifying a component only with respect to the *role* it plays in the greater whole. Clearly, a better approach would involve a combination of these two extremes (Fisher, 1986). We are investigating an extension to LABYRINTH in which the concept learner determines dynamically whether to classify an object based on its descriptive attributes only (as in the current system), or whether to consider its role as well.

The approach involves storing a *container* link with each object component. This link points from the component to the object of which it is a component (the partonomy parent) and is treated as an additional attribute for that object. Handa (1990) has explored one version of this approach. His system extends LABYRINTH to learn context-sensitive concepts by classifying each component twice: first to get a label used in classifying its container, and again after the container has been classified, using the container link as an additional attribute. In contrast, we plan to use ideas from Gennari's (1989) model of selective attention to determine dynamically whether to use the container attribute or others in classifying the object.

Another interesting extension to LABYRINTH involves forming concepts for the *roles* in its hierarchy. Consider the LEFTSTACK concept in Figure 3. The two values in COMPONENT_2 (ODD-1 and CIRCLE-1) are grouped by a simple kind of *functional* similarity; they play the same *role* in a structured concept. This grouping might occur in several structured concepts (although not in the STACK domain); however, the

current system has no means of recognizing this similarity across roles. We plan to investigate mechanisms through which Labyrinth could recognize such shared structures.

5.3 Domains for Labyrinth

We claim in Section 3.1 that a hierarchical organization is natural for many domains; we plan to demonstrate Labyrinth's effectiveness in such domains. We have designed Labyrinth as the fundamental memory organization scheme for Icarus (Langley, Thompson, Gennari, Iba, & Allen, in press), an integrated architecture that treats storage and retrieval as central issues. Two components of Icarus use structured object descriptions, and we plan to integrate each of these other systems with Labyrinth.

Dædalus (Langley & Allen, 1990), the planning component of the architecture, uses plan knowledge stored in a probabilistic concept hierarchy to guide operator selection. We are currently integrating Labyrinth into Daedalus; in this domain, a means-ends trace is represented as a hierarchical object linked by a Subgoal relation. In addition, we plan to apply Labyrinth to the motor schemas formed by Mæander (Iba & Gennari, this volume), which represents limb motions as temporal sequences of joint positions and velocities. Each state corresponds to a Labyrinth component, with joints serving as primitive objects.

6. Summary

In this chapter, we have described a system that learns concepts in structured domains. We have explained why the study of structured domains is important, and we have described six related systems that form a historical background for the current work. We have emphasized that all of the characteristics of Labyrinth have been found in at least one of these systems. However, no single system shares all five features: the use of probabilistic concepts; an incremental algorithm; learning from unclassified instances; learning with objects that have relations; and using component structure to constrain matching.

Labyrinth is an implemented system that extends Cobweb to structured domains. The system demonstrates a method for learning from hierarchically decomposed objects, using the results of component clas-

sifications to guide object classification. It learns in the presence of arbitrary relations in the object and concept language. LABYRINTH also introduces a new method for learning with hierarchically structured attributes. It demonstrates a form of representation change, in that it not only forms new terms as in previous concept formation systems, but also *uses* those terms in describing new concepts. In future work, we hope to establish LABYRINTH's applicability in a wide range of domains and to test its abilities in systematic experiments.

Acknowledgements

We thank Kathleen McKusick, Wayne Iba, Deepak Kulkarni, John Gennari, John Allen, and Doug Fisher for lengthy discussions that have influenced many of the ideas in this paper. All of the above and Sally Mouzon provided useful comments on an earlier draft.

References

Allen, J. A., & Langley, P. (1990). Integrating memory and search in planning. *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control* (pp. 301–312). San Diego, CA: Morgan Kaufmann.

Dietterich, T. G. (1986). Learning at the knowledge level. *Machine Learning*, *1*, 287–316.

Dietterich, T. G., & Michalski, R. S. (1981). Inductive learning of structural descriptions. *Artificial Intelligence*, *16*, 257–294.

Everitt, B. (1981). *Cluster analysis*. London: Heinemann.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1989). The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, *41*, 1–63.

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. (1986). A proposed method of conceptual clustering for structured and decomposable objects. In T. M. Mitchell, J. G. Carbonell, & R. S. Michalski (Eds.), *Machine learning: A guide to current research*. Boston: Kluwer.

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

Fisher, D. H., & Langley, P. (1990). 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. (1989). Focused concept formation. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 379–382). Ithaca, NY: Morgan Kaufmann.

Gennari, J. H., Langley, P., & Fisher, D. (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.

Handa, K. (1990). Cfix: Concept formation by interaction of related objects. *Proceedings of the Pacific Rim International Conference on Artificial Intelligence*. Nagoya, Japan.

Hanson, S. J., & Bauer, M. (1989). Conceptual clustering, categorization, and polymorphy. *Machine Learning*, *3*, 343–372.

Hayes-Roth, F., & McDermott, J. (1978). An interference matching technique for inducing abstractions. *Communications of the ACM*, *21*, 401–410.

Hayes-Roth, F. (1978). The role of partial and best matches in knowledge systems. In D. A. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press.

Hoff, W., Michalski, R. S., & Stepp, R. E. (1983). *A program for learning structural descriptions from examples* (Tech. Rep. No. UIUCDCS-F-83-904). Urbana: University of Illinois, Department of Computer Science.

Kline, P. J. (1981). The superiority of relative criteria in partial matching and generalization. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 296–303). Vancouver, BC: Morgan Kaufmann.

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

Langley, P., Thompson, K., Iba, W., Gennari, J. H., & Allen, J. A. (in press). An integrated cognitive architecture for autonomous agents. In W. Van De Velde (Ed.), *Representation and learning in autonomous agents*. Amsterdam: North Holland.

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

Levinson, R. A. (1985). *A self-organizing retrieval system for graphs*. Doctoral dissertation, Department of Computer Sciences, University of Texas, Austin.

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. San Francisco: W. H. Freeman.

McNamara, T. P., Hardy, J. K., & Hirtle, S. C. (1989). Subjective hierarchies in spatial memory. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *15*, 211–227.

Mervis, C., & Rosch, E. (1981). Categorization of natural objects. *Annual Review of Psychology*, *32*, 89–115.

McKusick, K. B., & Thompson, K. (1990). *COBWEB/3: A portable implementation* (Tech. Rep. No. FIA-90-6-18-2). Moffett Field, CA: NASA Ames Research Center, Artificial Intelligence Research Branch.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

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

Mitchell, T. M., Utgoff, P., & Banerji, R. B. (1983). Learning problem solving heuristics by experimentation. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

Nordhausen, B., & Langley, P. (1990). An integrated approach to empirical discovery. In J. Shrager & P. Langley (Eds.), *Computational models of scientific discovery and theory formation*. San Mateo, CA: Morgan Kaufmann.

Papademetriou, C., & Steiglitz, K. (1982). *Combinatorial optimization*. Englewood Cliffs, NJ: Prentice Hall.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Rubin, J. M., & Richards, W. A. (1985). *Boundaries of visual motion* (AI Memo 835). Cambridge, MA: Massachusetts Institute of Technology, Laboratory for Artificial Intelligence.

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

Stepp, R. E. (1984). *Conjunctive conceptual clustering: A methodology and experimentation*. Doctoral dissertation, Department of Computer Science, University of Illinois, Urbana.

Stepp, R. E., & Michalski, R. S. (1986). Conceptual clustering of structured objects: A goal-oriented approach. *Artificial Intelligence, 28*, 43–69.

Vere, S. A. (1975). Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 281–287). Tbilisi, USSR: Morgan Kaufmann.

Wasserman, K. (1985). *Unifying representation and generalization: Understanding hierarchically structured objects*. Doctoral dissertation, Department of Computer Science, Columbia University, New York.

Watanabe, L., & Rendell, L. (1990). Effective generalization of relational descriptions. *Proceedings of the Eighth National Conference of the American Association for Artificial Intelligence* (pp. 875–881). Boston, MA: AAAI Press.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The psychology of computer vision*. New York: McGraw-Hill.

Wogulis, J., & Langley, P. (1989). Improving efficiency by learning intermediate concepts. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 657–662). Detroit, MI: Morgan Kaufmann.