DISCOVERING PATTERNS IN SEQUENCES OF OBJECTS

by

Tom Dietterich
Ryszard S. Michalski

Proceedings of the International Machine Learning Workshop, June 22–24, 1983.

DISCOVERING PATTERNS IN SEQUENCES OF OBJECTS

Thomas G. Dietterich
Department of Computer Science
Stanford University
Stanford, CA 94305

Ryszard S. Michalski
Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

A more general kind of sequence-prediction problem—the non-deterministic prediction problem—is defined, and a general methodology for its solution presented. The methodology, called SPARC, employs multiple description models to guide the search for plausible sequence-generating rules. Three different models are presented along with algorithms for instantiating them to discover rules. The instantiation process requires that the initial input sequence be substantially transformed to make explicit important features of the sequence. Four different data transformation operators are described. The architecture of a system called SPARC/E is presented, which implements most of the methodology for discovering sequence-generating rules in the card game Eleusis. Examples of the execution of SPARC/E are presented.

1. Introduction

Inductive learning—that is, learning by generalizing specific facts or observations—is a fundamental strategy by which we acquire knowledge about the world. This form of learning is rapidly becoming one of the central research topics in AI. Most research on computer models of inductive learning has addressed the problem of inducing a general description of a concept from a collection of independent instances of that concept (the so-called training instances). Thus, the research has dealt with learning concepts that represent a certain class of instances. The instances can be specific physical objects, interactions, actions, processes, and so on. The learned concepts are general descriptions of classes of such instances.

Learning problems of this type include

- learning a checkers evaluation function [Samuel, 1963, 1967] that assigns to a given class of board situations a certain value,
- learning descriptions of block structures [Winston, 1970],
- determining rules for interpreting mass spectrograms [Buchanan and Mitchell, 1978],
- formulating diagnostic rules for soybean diseases [Michalski and Chilausky, 1980], and
- discovering heuristics to guide the application of symbolic integration operators [Mitchell, Utgoff, and Banerji, 1983].

In Samuel's checkers program, for example, each training instance was a board situation represented as a vector of 16 attributes. The learned concept was an evaluation function that computed the "value" of any board position for the side whose turn it was to move. No relationships between different board positions were considered. Similarly, Michalski's AQ11 program [Michalski and Chilausky, 1980] was given independent training instances, each describing a diseased soybean plant in terms of 35 multi-valued attributes. Each plant could have one of 19 possible soybean diseases. From several hundred training instances, the program inferred general diagnostic rules for each of these diseases.

This type of inductive learning can be called *instance-to-class generalization*. A review of several methods for such instance-to-class generalization can be found in [Michalski, Carbonell, and Mitchell, 1983]. A comprehensive review of learning research is given in [Dietterich, London, Clarkson, and Dromey, 1982].

Another type of inductive learning involves constructing a description of a whole object by observing only selected parts of it. For example, given a set of fragments of a scene, the problem is to hypothesize the description of the whole scene. A very important case of such *part-to-whole generalization* is where the "part" consists of a fragment of a sequence of objects (or a process evolving in time) and the problem is to induce the hypothetical description of the whole sequence (the process). Once such a description is found, it can be used to predict the possible continuations of the given sequence or process. This class of part-to-whole inductive learning problems we will call *prediction problems*.

This paper investigates the prediction problem for a sequence of objects characterized by a finite set of attributes. An elementary problem of this type is letter-sequence prediction, in which each object in the sequence is characterized by only one attribute: the name of the letter. For example, given a sequence of letters such as

A B X B C W C D V ...

the learning program must discover a "pattern"—that is, a rule that governs the generation of letters in the sequence. In this case, such a rule might state that the sequence is a periodically repeating subsequence of three letters in which the first two letters are successors of the letter appearing in the previous period, while the third letter is the predecessor of the corresponding letter in the previous period. Early papers by Simon and Kotovsky [1963, 1972, 1973] show that just a few relationships (such as successor, predecessor, and equality) are sufficient to represent most such patterns. Related work by Solomonoff [1964] and Hedrick [1976] has investigated grammatical approaches to describing letter sequences.

The sequence prediction problem becomes more difficult when the sequence consists not of simple objects with only a single relevant attribute (like the problem just described), but instead of objects with many relevant attributes. Further complexity is introduced if the pattern describing the sequence also involves a variety of relationships among these attributes. For example, the pattern may involve the periodicity of recurrence of certain properties or the dependence of the next object in the sequence on the properties of objects preceding

The authors gratefully acknowledge the partial support of the NSF under grant MCS-82-05166 and of the Office of Naval Research under grant No. N00014-82-K-0186.

it at some arbitrary distance in the past. A sequence prediction problem exhibiting the above-mentioned complexities arises in the card game Eleusis [Abbott, 1977; Gardner, 1977]. Examples from this game will be used to illustrate the general methodology of discovering patterns in sequences described in this paper. The rules for Eleusis are briefly explained in section 2.1.

Before we formulate precisely this problem of discovering patterns in sequences, let us first explain why it is important for current AI research. There are three major AI problems that must be addressed in any solution to this discovery task: (a) the representation problem, (b) the problem of performing model-driven inductive learning with multiple models, and (c) the problem of reasoning about temporal processes. The specific representation problem of interest here is that of automatically determining an appropriate series of transformations of the initial sequence description so that the pattern can be found. The multiple-model inductive learning problem arises because no single model can provide sufficient guidance to the search for plausible descriptions in this domain. The relationship of this problem to reasoning about time is not as strong as the other two problems. However, since temporal processes include as a special case discrete-time linear sequences, some of the techniques developed for sequence prediction may be relevant to the more general problem.

In the next two sections, we discuss in detail the representation problem and the problem of multiple-model induction as they arise in this domain.

1.1. Task-oriented transformation of description space

The problem of transforming the initial problem description arises in many practical domains in which the given data (e.g., the training instances in inductive learning) are observations or measurements that do not include the information most relevant to the task at hand. For example, in character recognition, the input typically consists of a matrix of light intensities representing a character, but the relevant information includes position-invariant properties of letters such as the presence of a line on the left or right of a character, occurrence of line endings, closed contours, and so on (e.g.,[Karpinski and Michalski, 1966]). These position-invariant properties can be made explicit by applying *task-oriented* transformations to the raw data.

An example of a learning program that performs task-oriented transformations is INTSUM (a part of the Meta-DENDRAL system, [Buchanan and Mitchell, 1978]). INTSUM is presented with raw training instances in the form of chemical structures (graphs) and associated mass spectra (represented as fragment masses and their intensities). For each fragment in the mass spectrum, INTSUM must determine the bonds that could have broken to produce that fragment. A simple mass spectrometer simulator is used to develop these hypothesized bond breaks. Each of the resulting transformed training instances has the form of a chemical structure and a set of bonds that broke when that structure was placed in the mass spectrometer. It is this information that is provided to the remaining parts of the Meta-DENDRAL system (programs RULEGEN and RULEMOD).

In character recognition programs and in Meta-DENDRAL, the data transformations are fixed in advance. Future learning systems, however, may not know the proper transformations *a priori*. These learning systems will need to select or invent appropriate task-oriented transformations for each learning situation.

This description-space transformation problem has been called by various authors the *data interpretation problem* [Dietterich, et al., 1982] or the *reformulation problem* [Amarel, 1968]. We prefer the term *task-oriented transformation problem*, since it emphasizes that the proper choice of data transformations depends upon the task being performed. In the sequence prediction problem discussed in this paper, the desired sequence-generating rules are described in a language quite different from the language used to describe the raw sequence. The learning system determines appropriate data transformations from four general classes of transformations and applies them to the raw sequence to produce a transformed sequence amenable to pattern discovery.

The task-oriented transformation problem is part of a spectrum of problems faced by learning programs. The simplest learning algorithms (e.g., linear regression) determine the coefficients for a predetermined, fixed set of variables. Slightly more sophisticated are learning algorithms, such as the A^q algorithm [Michalski and Kulpa, 1971] or the candidate elimination algorithm [Mitchell, 1978], that are able to determine which terms are relevant and how they should be combined (i.e., with operators such as $\wedge$ and $\vee$). Learning algorithms that perform interpretative transformations (e.g., Soloway [1981], Meta-DENDRAL [Buchanan and Mitchell, 1978]) augment these basic inductive algorithms by applying a set of predetermined transformations to the data prior to inductive generalization. Not yet developed are learning algorithms that could select description-space transformations under guidance of special heuristics. And very few researchers have addressed the problem of discovering new descriptors (predicates, functions, operators, etc.). Table 1-1 shows this spectrum of inductive learning problems.

1. Determine coefficients
2. Select relevant variables and combine
3. Apply predetermined transformations
4. Select transformations under heuristic guidance
5. Discover new descriptors

Table 1-1: Spectrum of learning problems in increasing order of difficulty

The method presented in this paper falls under category 4, since it searches four general classes of transformations and employs heuristics reflecting domain-specific knowledge.

1.2. Learning with multiple models

The second major problem that arises in sequence prediction is the problem of learning using multiple description models. This problem has not received much attention in previous AI research. Most existing systems employ a single model that provides guidance to the induction algorithm as it searches a space of possible descriptions. Many systems, for example, use conjunctive descriptions to represent concepts. By constraining the search to consider only conjunctive descriptions, the learning problem is greatly simplified. Michalski [Michalski and Kulpa, 1971] constrains descriptions to be in disjunctive normal form with *fewest* disjunctive terms. This constraint is satisfied (approximately) by having the induction algorithm find first one conjunction, and then another, and so on until all of the training instances are covered. Meta-DENDRAL [Buchanan and Mitchell, 1978] employs a fairly elaborate model of the operation of the mass spectrometer to guide its search for cleavage rules. In general, all of these systems use a single model, and very few authors have made their models explicit.

One researcher who has employed multiple models is Persson [1966]. He applied four different models to the problem of extrapolating number- and letter-sequences. Briefly, these models were

1. a model that computes the coefficients and the degree of a polynomial by applying Newton's forward-difference formula (the degree can be arbitrarily large);

2. an extended model that discovers exponential rules of the form AB^C, where A is a polynomial of degree 4 or less and B and C are polynomials of degree 1 or less (i.e., B and C are of the form $ax + b$);

3. a simple periodic model for periods of length 2 (i.e., intertwined sequences); and

4. a generalization of the Kotovsky and Simon model for Thurstone letter-series that can discover simple periodic and segmented sequence-generating laws.

These models are applied in an artificial learning situation in which the program is given a *sequence* of sequence-extrapolation problems. Thus, in addition to attempting to solve each individual sequence-extrapolation problem, Persson's program tries to predict the *kind* of sequence-prediction problem that it will next receive—that is, it tries to predict which model will best fit the next sequence-prediction problem. Hence, when the program is attempting to solve one of the base-level problems, it selects models to apply based on its predictions about the kind of sequence that it is expecting.

Persson's work shows the value of employing multiple description models to search for sequence-generating rules. The major limitation of Persson's approach, however, is that it is specific to number- and letter-sequence prediction. His methods cannot solve the more general problem described in this paper in which objects have multiple attributes and the task is to find a nondeterministic sequence-prediction rule.

Table 1-2 shows a spectrum of five model-based learning methods. The simplest approach is to use a single fixed model. This has been the common approach in AI thus far. The next step is to provide a learning program with a set of models from which it would choose the most appropriate ones. This is the approach used by Persson. The third level of sophistication would be to have the program generate a predetermined set of models, just as the learning program applies a predetermined set of data transformations. This could be improved by having the program decide which models to generate on the basis of special heuristics. Finally, an even more sophisticated program would be able to invent new models and apply them to guide the learning process.

1. Single model
2. Selection from a few models
3. Predetermined generation of models
4. Heuristically-guided generation of models
5. Discovery of new models

Table 1-2: Spectrum of model-based methods in increasing difficulty

The approach described in this paper searches a predetermined space of possible models in a depth-first fashion, and hence, falls under point 3 of this table.

It is the development of techniques for addressing these two problems—of selecting task-oriented transformations and of applying multiple description models—that is the main theoretical contribution of this research. In the remainder of this paper, we

1. define the sequence prediction problem under consideration,

2. describe the methods used for representing and transforming the initial training instances,

3. present techniques for representing the models and sequence-generating rules, and finally,

4. provide the details of the program SPARC/E, which implements most of the described methodology. The program is illustrated by a few selected examples of its operation when applied to the inductive card game Eleusis.

2. Problem Statement

Suppose we are observing a process that generates some objects, one after another, and arranges them into a sequence. Suppose that the objects are generated from a known set and that there exists an underlying law that specifies at least some of the properties of every new generated object. We will call such a law a *sequence-generating rule.* It is assumed that the law is expressed in terms of properties that are either observable properties of objects present in the sequence up to the moment when a new object is generated or properties that can be derived from such observable properties by some known inference rules.

We are interested in the most general kind of sequence-generating law in which the law does not necessarily completely determine which objects can or cannot appear next in the sequence. The law merely states some properties that constrain the next object to be a member of a restricted set. Thus, such a generating rule is nondeterministic. The task of discovering such a generating law is a difficult learning task, requiring task-specific data transformations and model-guided induction. We will call this learning problem a *non-deterministic prediction problem* (NDP, for short). If the law guiding the generation of the sequence completely defines the next object at every point in the sequence, then the NDP problem reduces to a *deterministic prediction problem* (DP, for short). In the DP problem, it is assumed that there is no randomness in the generation of the next object. The next object is strictly a function of the past objects.

Many researchers have previously considered DP problems such as letter-sequence prediction, number-series extrapolation, economical prediction, and prediction of the behavior of a computer system. Most recently, the BACON system [Langley, 1980] has addressed a wide range of DP problems that arise in scientific discovery situations. BACON and most of its predecessors make strong use of the constraint that in a DP problem, *all* attributes of the next object in the sequence are determined by the previous objects in the sequence. The NDP problem is more difficult to solve, because only a partial description of the original sequence is sought. Consequently, many more plausible hypotheses must be considered during the inductive learning process.

Let us illustrate a simple NDP problem by an example. Suppose we are given a snapshot of an ongoing process that has already generated the objects (graphs) shown in Figure 2-1.

The observable properties of each graph are: the NUMBER OF NODES, the SHAPE of the graph (T-junction, square, bar, wheel,

triangle, star, diamond), the TEXTURE of each node (solid black, blank, and cross), and the ORIENTATION of the graph (applicable only to graphs that are elongated in some direction, expressed as degrees clockwise from vertical). Suppose we would like now to predict what could be the next object.

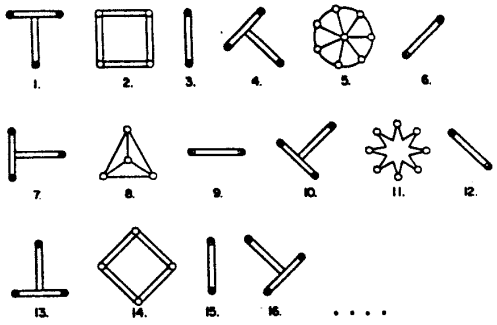

Figure 2-1: A simple NDP problem

By examining the given string in Figure 2-1, we can observe that it can be partitioned into segments of three graphs in length. The nodes of the graphs in each triplet have TEXTURE in the order <solid black, blank, cross>. The SHAPES of the graphs are always <T-junction, *, bar> (where * denotes any shape). We can also notice that the ORIENTATION of the T-junction changes by -45 degrees each time, while the ORIENTATION of the bar increases by $+45$ degrees each time. Finally, the NUMBER OF NODES in the center graph alternates between 4 and 8. If the above regularities indeed constitute the generating law, we can hypothesize that the next graph in the sequence will have 8 blank nodes, and then after that there will be a graph that is a slanted bar with crossed nodes and ORIENTATION of 225 degrees (slant downwards to left). Thus, with regard to the first predicted object, we know only two properties (NUMBER and TEXTURE of nodes), and with regard to the second predicted object, we know it completely. It is easy to see that the problem of letter-sequence prediction (or extrapolation) is a special case of the NDP problem where each object is a letter of an alphabet whose observable property is its name. It also has one derived property that is its position in the alphabet. (The order of letters in the alphabet is externally-provided domain knowledge.) Since each object (in this case a character) is defined completely by specifying its name (or its position in the alphabet), letter series prediction is necessarily a DP problem.

2.1. An exemplary NDP problem: the card game Eleusis

An interesting NDP problem occurs in the card game Eleusis, invented by Robert Abbott [Abbott, 1977; Gardner, 1977]. Eleusis is an inductive game in which players attempt to discover a "secret rule" invented by the dealer. The secret rule is the generating rule for a sequence of cards. Each player, in his or her turn, adds one card to the sequence, and the dealer indicates whether the card is a correct extension of the sequence (i.e., satisfies the secret sequence-generating rule). Players who play incorrectly are penalized by having additional cards added to their hands. The goal of each player is to get rid of all of the cards in his hand, which is only possible if correct cards are played. The cards played during the game are displayed in the form of a layout in which the correct cards form a "main line" and incorrect cards form "side lines" branching down from the main line at the card that they followed. Figure 2-2 shows a typical Eleusis layout for the sequence-generating rule "Play alternating red and black cards." In this game, the 3 of hearts was played first, followed by a 9 of spades, and a Jack of diamonds. All of these were correct. Following the Jack, a 5 of diamonds was played. It appears on a sideline below the Jack, because it was not a correct extension of the sequence. (At this point a black card is required.) The 4 of clubs was then correctly played, and so on.

Main line:	3H	9S	JD	4C	JD	2C	10D	2C	6H
Side lines:			5D		AH	AS	8H		
					8H	10S	7H		
					QD		10H		

Figure 2-2: A sample Eleusis layout

Eleusis provides a good domain for studying the use of task-oriented data transformations to aid learning. Frequently, the generating law for an Eleusis sequence is stated in terms of descriptors that are not present in the initial sequence. In this example, for instance, the generating law is stated in terms of the color of the cards, but the original sequence supplies only the RANK and SUIT of each card. Table 2-1 provides some examples of generating laws from Eleusis. Note that the terms in which these laws are expressed (e.g., "strings of cards of the same suit", "alternating sequence") are quite different from terms such as RANK and SUIT that described the original sequence. To bridge this difference, appropriate description-space transformations have to be performed.

- If the last card was a spade, play a heart; if last card was a heart, play diamonds; if last was diamond, play clubs; and if last was club, play spades.
- The card played must be one point higher than or one point lower than the last card.
- If the last card was black, play a card higher than or equal to that card; if the last card was red, play lower or equal.
- Play alternating even and odd cards.
- Play strings of cards such that each string contains cards all in the same suit and has an odd number of cards in it.

Table 2-1: Some examples of sequence-generating rules in Eleusis

Eleusis also provides a good domain for studying the use of models for guiding the induction process. The space of possible Eleusis rules using descriptors such as SUIT, RANK, COLOR, FACEDNESS, PARITY, PRIMENESS, and RANK MODULO 3 is very large. In our description language, there are more than 10^{137} possible sequence-generating rules involving four or fewer conjunctive expressions[1] A breadth-first search of this space, such as is conducted by the candidate-elimination

[1] This estimate is based on computing the space of all syntactically legal VL1 conjuncts containing the following set of descriptors (after each descriptor is listed the number of elements in its value set and the number of possible selectors that can be formed using those elements): SUIT (4,9), RANK (13, 91), COLOR (2,3), FACEDNESS (2,3), PARITY (2,3), PRIMENESS (2,3), RANKMOD3 (3,7), D-SUIT01 (4,9), D-SUIT02 (4,9), D-RANK01 (25,300), D-RANK02 (25,300), S-RANK01 (25,300), S-RANK02 (25,300), D-COLOR01 (2,3),

algorithm, would clearly be impossible. Fortunately, the rules used by people tend to cluster into certain classes that can be well-described by three models: periodic rules, decomposition rules, and DNF rules. Thus, a model-directed approach can be used to discover sequence-generating rules in Eleusis.

3. Overview of Solution

This section gives an overview of the approach taken to solving the NDP problem defined in section 2. The approach is a combination of bottom-up data transformation, top-down model specialization, and data-driven instantiation of the specialized models to fit the transformed data. These three processes can be briefly explained as follows:

1. Bottom-up data transformation involves applying various transformation operators to the initial sequence description to obtain a *derived sequence description.* We use four basic data transformations: adding derived attributes, segmenting, splitting, and blocking. Details of these are described in section 4.

2. Top-down model specialization involves specifying particular values for the parameters of general rule models to obtain a *parameterized model.* We use three general models: the disjunctive normal form model (DNF), the decomposition model, and the periodic model. Each of these models has one or more parameters. For example, both the DNF and decomposition models have a single parameter: the *lookback*, L (i.e., the number of objects back from the given object in the sequence that are assumed to determine the next object). The periodic model has two parameters: the *lookback*, L, and the *period length*, P, which indicates the length of the repeating period in the sequence. Details of the model specialization process are described in section 5.

3. The model-instantiation process attempts to fit the parameterized model to the derived sequence description to produce an instantiated parameterized model. A model that has been parameterized and instantiated serves as a sequence-generating rule. This process is described in section 6.

The above three steps are illustrated schematically in figure 3-1.

Model instantiation, as used in this paper, is an extension of the well-known AI technique of schema instantiation. Schema instantiation has been applied, for example, by Schank and Abelson [1975] to interpret natural language, by Engelmore and Terry [1979] to interpret X-ray diffraction data in protein chemistry, and by Friedland [1979] to plan genetics experiments. Model instantiation differs from schema instantiation in the complexity of the instantiation process.

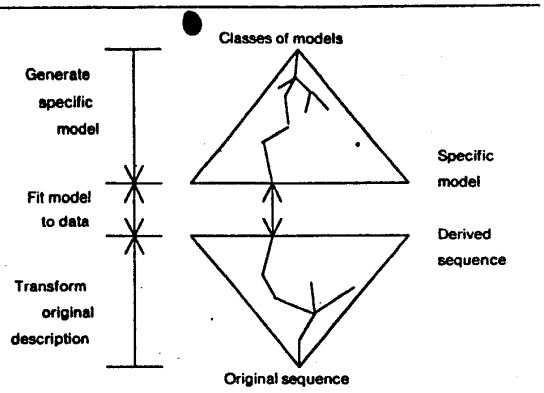

Figure 3-1: Schematic description of the rule discovery process

Model instantiation involves not only filling in predetermined slots or substituting constants for variables, but also synthesizing a logical formula of an assumed type. For example, in order to instantiate each of the three models described below, the program must synthesize a conjunction of predicates or a disjunction of such conjunctions that satisfies certain constraints. Model-instantiation methods share with schema-instantiation methods the advantage that they are efficient, and also effective with noisy and uncertain data. The constraints provided by the models (or schemas) drastically reduce the size of the space that the program must search.

The principal disadvantage of model- and schema-instantiation methods is that they require substantial amounts of domain knowledge to be built into the program. In order to keep this domain knowledge explicit and easily modified, we employ a ring architecture in the design of the learning program, as described in section 6. This architecture facilitates the application of the system to a variety of problems by simplifying the process of changing the domain-specific parts of the program.

4. Describing and Transforming Training Instances

Now that we have defined the problem to be solved (the NDP problem) and sketched the solution, we launch into the details of that solution. This section presents the description language for representing the original sequences and the transformation operators that can be applied to modify that representation.

4.1. Representing the initial sequence

A sequence of objects is represented as an indexed sequence[2]

$$\langle q_1, q_2, ..., q_k \rangle$$

It is assumed that the only relevant relationship between two objects is their ordering in the sequence. Each object is described by a set of attributes (also called *descriptors*) $f_1, f_2, ..., f_n$, which can be viewed as functions mapping objects into attribute values. To state that attribute f_i of object q_j has value r, we write

$$[f_i(q_j)=r].$$

D-COLOR02 (2,3), D-FACEDNESS01 (2,3), D-FACEDNESS02 (2,3), D-PARITY01 (2,3), D-PARITY02 (2,3), D-PRIMENESS01 (2,3), D-PRIMENESS02 (2,3), D-RANKMOD3-01 (3,7), D-RANKMOD3-02 (3,7). The SUIT and RANKMOD3 descriptors are cyclically ordered, while the RANK descriptors are interval descriptors. All others are nominal. In a block of three adjacent cards (with lookback L=2), the first seven descriptors appear three times—once for each card. Hence, the total number of possible conjuncts is $(9*91*3*3*3*3*7)^3 * (9*300*300*3*3*3*7)^2 = 2.1122\text{I}*10^{34}$. If there are four conjuncts in a rule, then we obtain $[2.11221*10^{34}]^4 = 1.99*10^{137}$.

[2] A summary of the notational conventions used in this paper appears in section 9.

This notation is called a *selector*. For example, if f_i is *color* and r is *red*, then the selector

`[color(qj)=red]`

states that the color of the j-th object in the sequence is red.

Each attribute is only permitted to take on values from a finite value set called the *domain*, $D(f_i)$, of that attribute. This constraint is part of the background knowledge that has to be given to the program. For example, in a deck of cards, the domain of the SUIT attribute is {clubs, diamonds, hearts, spades}. Additional knowledge about the domain set can be represented. In particular, the domain set may be linearly ordered, cyclically ordered (i.e., in a circular, wrap-around ordering), or tree ordered. We will see below how these domain orderings are applied to the problem of representing cards in an Eleusis game.

A complete initial description of a single object, q_j, called an *event*, is an expression giving the values for all of the attributes of q_j. This is usually written as a conjunction of selectors:

`[f1(qj)=r1][f2(qj)=r2]...[fn(qj)=rn].`

It can also be represented as a vector of attribute values:

$$(r_1, r_2, ..., r_n).$$

This vector notation suggests that each object description can be viewed as a point in the *event space* E:

$$E = D(f_1) \times D(f_2) \times ... \times D(f_n)$$

This event space contains all possible events.

A complete description of the initial sequence is a sequence of conjunctions of selectors (or alternatively, a sequence of attribute vectors)—one conjunction for each object in the sequence. The space of all possible sequences can be generated by selecting all possible sequences of events chosen from E.

4.2. Transforming the Sequence

As we mentioned in section 1, it is often necessary to transform the initial sequence into a *derived sequence* in order to facilitate the discovery of sequence-generating rules. Such a data transformation can be viewed as a mapping T from one set of sequences S, containing objects Q, described by attributes F, to another set of *derived sequences* S', containing *derived objects* Q', and described by *derived attributes* F'.

$$T_{p_1, ..., p_k} : \langle S, Q, F \rangle \dashrightarrow \langle S', Q', F' \rangle$$

where p_1, ..., p_k are parameters of the transformation that control its application. We have found four basic transformations to be especially useful for discovering sequence-generating rules: (a) *adding derived attributes*, (b) *segmenting*, (c) *splitting into phases*, and (d) *blocking*. Each of these is described in turn.

4.2.1. Adding derived attributes

The simplest transformation does not change the set of sequences, S, or the set of objects, Q, but only the set of attributes, F. For example, in Eleusis, the initial set F contains only two attributes: the RANK and SUIT of a card. These can be augmented by deriving such attributes as COLOR (red or black), FACEDNESS (faced or nonfaced), PARITY (odd or even), and PRIMENESS (prime or not prime in rank). The adding-derived-attributes transformation has no parameters.

4.2.2. Segmenting

The segmenting transformation derives a new sequence made up of a new set of objects, Q', and described with a new set of attributes, F'. The new sequence is produced from the original sequence by dividing the original sequence into non-overlapping segments. Each segment becomes a derived object in the new sequence. The only parameter of the segmenting transformation is the segmentation condition that tells how the original sequence should be divided into segments. Three types of segmentation conditions can be distinguished: (a) those that use properties of the *original objects* to determine where the sequence *should* be broken, (b) those that use properties of the *original objects* to determine where the sequence should *not* be broken, and (c) those that use properties of *derived objects* to determine where the original sequence *should* be broken.

For example, suppose the original sequence consists of physical objects described by attributes such as WEIGHT, COLOR, and HEIGHT. An example of each type of segmentation condition follows:

1. Break when `[weight(qi-1)>10][weight(qi)≤10]`.

   According to this condition, the original sequence is to be broken (between q_{i-1} and q_i) at the point where the weight of an object changes from above 10 to under 10.

2. Don't break as long as `[color(qi)=color(qi-1)][weight(qi)>10]`.

   This condition states that the original sequence will not be broken (between q_{i-1} and q_i) if the color stays the same and the weight remains above 10. It will be broken at any point where these conditions do not both hold.

3. Break so that `[length(qi')=2]`.

   This condition states that derived objects (q_i') should be subsequences of length 2 from the original sequence (i.e., pairs of adjacent objects from the original sequence).

The choice of attributes, F', for describing the newly-derived objects, Q', depends on the segmentation condition used to segment the sequence. For example, if the `[length(qi')=2]` condition is used, attributes of interest might include the sum of the VALUES of the two original objects, the maximum VALUE, the minimum VALUE, and so on. The LENGTH of the segment would not be of interest, since by definition, it is a constant. However, if the `[color(qi)=color(qi-1)]` condition is used, the LENGTH of the segment could be quite interesting and should be derived. Also, the COLOR shared by all of the cards in the segment might be of interest. In our implementation, the user specifies which attributes should be derived. All user-specified attributes are derived unless the program can prove from the segmentation condition that those attributes would not have a well-defined value for each segment in the sequence or else would be trivially constant for all segments.

Often, a segmentation condition leads to the creation of incomplete segments at the beginning and end of the original sequence. These boundary cases can create difficulties during model instantiation, so they are ignored during rule discovery, but checked during rule evaluation.

4.2.3. Splitting

The splitting transformation splits a single sequence into a *sequence* of P separate subsequences: $\langle ph_1, ph_2, ... \rangle$. Sequence ph_i starts with object q_i (the object at the i-th position in the original sequence) and continues with objects taken from succeeding positions at distance P apart in the original sequence. Each of the derived sequences is called a *phase*. P is the parameter of the splitting transformation that denotes the number of phases. Figure 4-1 shows the splitting operation for P = 3.

```
Original sequence: <q1  q2  q3  q4  q5  q6  q7  q8  q9>
Derived sequence: <ph1 ph2 ph3>, where
     ph1:   <q1          q4          q7>
     ph2:     <q2          q5          q8>
     ph3:       <q3          q6          q9>
```

Figure 4-1: Splitting transformation with P=3

The objects within each phase retain the linear ordering that they had in the original sequence. The phases themselves can be considered to be cyclically ordered so that ph_1 precedes ph_2, which precedes ph_3, and so on, until ph_P, which is followed by ph_1 again. Consider, for example, the following sequence:

<1 8 2 9 3 10 4 11>

The splitting transformation with P=2 would produce the sequence <ph1 ph2> where

ph1 = <1 2 3 4>
ph2 = <8 9 10 11>

Since the splitting transformation simply breaks the original sequence of objects into subsequences, no new objects are created. Furthermore, no new descriptors are defined. The set of descriptors used to characterize the objects in each of the phases is the same as the set of descriptors used to characterize the objects in the original sequence.

The splitting transformation can be applied to break one sequence-prediction problem into several subproblems—one for each phase. This is how periodic rules are discovered.

4.2.4. Blocking

The blocking transformation converts the original sequence into a new sequence made up of a new set of objects B' and a new set of attributes F'. The new sequence is created by breaking the original sequence into overlapping segments called *blocks*. Each object b_i in the new sequence describes a block of L+1 consecutive objects from the original sequence, starting at object q_i (called the *head*) and proceeding backwards to object q_{i-L} (where L is the lookback parameter of the blocking transformation). Figure 4-2 shows the blocking operation for L=2 (Block length of 3).

Several attributes are derived to describe each block. For each attribute A applicable to the objects in the original sequence, the attributes A0, A1, ..., AL are defined that are applicable to the objects in the derived sequence. $A0(b_i)$ has the same value as $A(q_i)$; $A1(b_i)$ has the same value as $A(q_{i-1})$; and so on until $AL(b_i)$, which has the same value as $A(q_{i-L})$. In other words, the original attributes are retained in the new sequence, but they are renamed so that they apply to whole *blocks* rather than to individual objects in the original sequence. The numerical suffix on the new names encodes the relative position of the original object q_i in block b_j.

```
Original sequence:  <q1  q2  q3  q4  q5  q6  q7  q8>
Derived sequence:       <b3  b4  b5  b6  b7  b8>
where b_i are derived objects defined as follows:
     b3:   <q1  q2  q3>
     b4:     <q2  q3  q4>
     b5:       <q3  q4  q5>
     b6:         <q4  q5  q6>
     b7:           <q5  q6  q7>
     b8:             <q6  q7  q8>
```

The underlined object in each block is the head object.

Figure 4-2: The Blocking Transformation with L=2.

For example, suppose we have the sequence <q1 q2 q3 q4 q5> with attributes RANK and SUIT, where

```
[rank(q1)=2][suit(q1)=H]
[rank(q2)=4][suit(q2)=S]
[rank(q3)=6][suit(q3)=C]
[rank(q4)=8][suit(q4)=D]
[rank(q5)=10][suit(q5)=H]
```

Now suppose we apply the blocking transformation to this sequence with L=2 to obtain the derived sequence of blocks <b3 b4 b5>. Then the descriptors RANK0, RANK1, RANK2, SUIT0, SUIT1, and SUIT2 will be derived with the values

```
[rank2(b3)=2][suit2(b3)=H]
    [rank1(b3)=4][suit1(b3)=S]
    [rank0(b3)=6][suit0(b3)=C]
[rank2(b4)=4][suit2(b4)=S]
    [rank1(b4)=6][suit1(b4)=C]
    [rank0(b4)=8][suit0(b4)=D]
[rank2(b5)=6][suit2(b5)=C]
    [rank1(b5)=8][suit1(b5)=D]
   [rank0(b5)=10][suit0(b5)=H]
```

This transformation leads to a highly redundant representation of the information in the original sequence. For example, the information about SUIT and RANK of the original object q_3 is repeated as SUIT0 and RANK0 of block b_3, SUIT1 and RANK1 of block b_4, and SUIT2 and RANK2 of block b_5. However, this derived sequence of blocks facilitates the representation of the relationships between objects in the original sequence. Many sequence-prediction rules involve such relationships.

To represent relationships between objects, additional descriptors called *sum* and *difference* descriptors are defined. In the case of the above sequence, the descriptors S-RANK01, S-RANK02, D-RANK01, D-RANK02, D-SUIT01, and D-SUIT02 are created. The value of S-RANK01(b_i) is the sum of RANK0(b_i) and RANK1(b_i). The value of D-RANK01(b_i) is the difference between RANK0(b_i) and RANK1(b_i). Thus, in addition to the selectors shown above, the following selectors would also be derived for the new sequence:

```
[s-rank01(b3)=10][s-rank02(b3)=8]
    [d-rank01(b3)=2][d-rank02(b3)=4]
    [d-suit01(b3)=1][d-suit02(b3)=2]
[s-rank01(b4)=14][s-rank02(b4)=12]
    [d-rank01(b4)=2][d-rank02(b4)=4]
    [d-suit01(b4)=1][d-suit02(b4)=2]
[s-rank01(b5)=18][s-rank02(b5)=16]
    [d-rank01(b5)=2][d-rank02(b5)=4]
    [d-suit01(b5)=1][d-suit02(b5)=2]
```

From this representation, it is relatively easy to discover that [d-rank01(b1)=2] is true for all blocks b1.

Ordinarily, sum and difference attributes only make sense for attributes such as RANK whose domain sets are linearly ordered. We have extended the definition of difference to cover unordered and cyclically ordered domain sets as well. For an unordered attribute such as COLOR, whose domain set is {red, black}, D-COLOR01 takes on the value 0 if the COLOR0(b_i) = COLOR1(b_i) and 1 otherwise. For attributes with cyclically-ordered domain sets, such as SUIT (with values {clubs, diamonds, hearts, spades}), D-SUIT01 is equal to the number of steps in the forward direction that are required to get from SUIT1(b_i) to SUIT0(b_i). If SUIT1(b_i)=diamonds and SUIT0(b_i)=clubs, D-SUIT01(b_i)=3.

The sum and difference attributes make the ordering of the original sequence explicit in the attributes that describe each block.

Consequently, it is no longer necessary to represent the ordering between blocks. Hence, the model-fitting algorithms discussed below treat the derived sequence (of blocks) as an unordered set of events.

One difficulty with the above approach is that the numerical suffix notation is not very easy to read, especially when it is combined with a sum or difference prefix. Hence, we have developed an alternative representation that is more comprehensible. In this notation, selectors that refer to blocks, such as `[suit1(b1)=H]`, are written as selectors that refer to objects in the original sequence, such as `[suit(q1-1)=H]`. Similarly, selectors such as `[d-rank01(bi)=3]` are written as `[rank(q1)=rank(q1-1)+3]`. This notation makes the meaning of the selectors clear without having to explicitly mention the blocks b_i. For purposes of implementation, the first notation is better because it enables the program to treat all sequences—including derived sequences—uniformly. However, the second notation is more understandable and hence will be used for the rest of this paper.

5. Representing Sequence-generating Rules and Models

A *sequence-generating rule* is a function g that assigns to each sequence of objects, $\langle q_1, q_2, ..., q_k \rangle$, a non-empty set of *admissible next objects* Q_{k+1}:

$$g: \{\langle q_1, q_2, ..., q_k \rangle\} \rightarrow \{Q_{k+1}\}$$

Q_{k+1} is the set of all objects that could appear as the next object in the sequence. For example, in the rule "Play a card whose rank is one higher than the previous card", $g(\langle ... 4C \rangle) = Q_{k+1}$ is the set of cards {5C, 5D, 5H, 5S}.

The set Q_{k+1} may contain only one event, or it may contain a large set of possible events. If for all k, the sequence $\langle q_1, q_2, ..., q_k \rangle$ is mapped by g into a singleton set, then the rule is a *deterministic* rule; otherwise, it is a *nondeterministic* rule. This paper addresses the problem of discovering a nondeterministic sequence-generating rule, g, given the sequence $\langle q_1, q_2, ..., q_k \rangle$.

The sequence $\langle q_1, q_2, ..., q_k \rangle$ can be viewed as the set of assertions

$$q_1 \in g(\langle\rangle)$$
$$q_2 \in g(\langle q_1 \rangle)$$
$$\vdots$$
$$q_m \in g(\langle q_1, ..., q_{m-1} \rangle)$$

These assertions are positive instances of the desired sequence-generating rule.

In Eleusis, negative instances are provided by the cards on the sidelines—that is, the cards rejected by the dealer for being incorrect. A sideline card q_3^- played after card q_3 provides a negative instance of the form:

$$q_3^- \notin g(\langle q_1, q_2, q_3 \rangle)$$

The goal is to find a description for g that is consistent with these training instances and satisfies some preference criterion.

The preference criterion in our methodology (and in all learning systems) attempts to evaluate a candidate rule in terms of its generality, predictive power, simplicity, and so on. These semantic properties are difficult to compute, however. Instead, virtually all learning systems employ syntactic criteria that correspond in some way to these semantic criteria. Syntactic criteria—such as the number of selectors in a conjunction and the number of conjuncts in a disjunction—will only correspond to the semantic criteria if the representational framework is well chosen (See McCarthy [1958]). As we noted in the introduction, most previous AI research on learning has employed a single representational framework or model for describing the rules or concepts to be learned. In Eleusis, a single framework is insufficient. Instead, we have developed three basic models that were found to be useful: the DNF model, the decomposition model, and the periodic model. When these models are employed, syntactic criteria can be used to approximate semantic criteria during evaluation.

A *model* is a logical schema that specifies the syntactic form of a class of descriptions (in our case, sequence-generating rules). A model consists of *model parameters* and a set of *constraints* that the model places on the forms of descriptions. The process of specifying the values for the parameters of a model is called *parameterizing* the model. The process of filling in the form of the parameterized model is called *instantiating* the model. A fully-parameterized and fully-instantiated model forms a sequence-generating rule. Models can be instantiated using the original sequence, or, more typically, using a sequence derived by applying some of the data transformations discussed in the previous section.

All three models use the representation language VL22 as a building block for expressing sequence-generating rules. VL22 is an extension to the predicate calculus that uses the *selector* as its simplest kind of formula. The VL22 selector is substantially more expressive than the simple selector presented above in section 4.1. The simple selector has the form:

`[fi(qj)=r]`

whereas the VL22 selector has the form:

`[fi(x1, x2, ..., xn) = r1 v r2 v ... v rm]`

In the VL22 selector, attributes f_i can take any number of arguments $(x_1, x_2, ..., x_n)$. Furthermore, the attributes f_i can take on any one of a *set* of values $\{r_1, r_2, ..., r_m\}$. The **v** denotes the *internal disjunction* operator. Thus, the selector

`[rank(q1)=9 v 10 v J v Q v K]`

indicates that the rank of object q_i can be either 9, 10, J, Q, or K. The internal disjunction represents disjunction over the values of a single variable. In this case, it could be expressed alternatively as

`[rank(q1)≥9]`,

since the domain of the RANK attribute is known to be linearly ordered with a maximum value of K (King). To aid comprehensibility, VL22 provides the operators <, >, ≤, ≥, and ≠, in addition to the basic = operator.

Examples of typical selectors include:

`[rank(q1)≠rank(q1-1)]`
(paraphrase: the RANK of q_i is different from the RANK of q_{i-1})

`[suit(q1)=suit(q1-1)+1]`
(paraphrase: the SUIT increases by one from q_{i-1} to q_i)

`[rank(q1)+rank(q1-2)>10]`
(paraphrase: the sum of the RANKs of q_i and q_{i-2} is greater than 10)

Now that we have introduced the basic notation of VL22, each of the three rule models is presented in turn.

5.1. The DNF model

The DNF model supports the broad class of rules that can be expressed as a universally quantified VL22 statement in disjunctive

normal form. The DNF model has one parameter, the degree of lookback, L. An example of a DNF rule (with L=1) is:

```
∀i ([color(qi)=color(qi-1)] V
    [rank(qi)=rank(qi-1)])
```

In general, a DNF rule is a collection of conjuncts of the form

```
∀i (C1 V C2 V C3 V ... V Ck)
```

The universal quantification over i indicates that this description is true for all objects q_i in the sequence.

An additional constraint specified in the DNF model is that the number of conjuncts, k, should be close to the minimum that produces a description consistent with the data.

5.2. The Decomposition Model

The decomposition model constrains the description to be a set of implications of the form:

```
L1 => R1
L2 => R2
   .
   .
   .
Lm => Rm
```

where the => sign indicates logical implication.

The model states that the left- and right-hand sides, L_j and R_j, must all be VL22 conjunctions. The left-hand sides must be mutually exclusive and exhaustive—that is,

$$L_1 \vee L_2 \vee \ldots L_m \equiv \text{TRUE, and}$$
$$\forall j,k\ (j \neq k \Rightarrow (L_j \wedge L_k \equiv \text{FALSE})).$$

A decomposition rule describes the next object in the sequence in terms of characteristics of the previous objects in the sequence. For example, the rule

```
∀i([color(qi-1)=black]=>[parity(qi)=odd] V
[color(qi-1)=red]  =>[parity(qi)=even])
```

is a decomposition rule that says that if the last card was black, the next card must be odd, and if the last card was red, the next card must be even.

The decomposition model has a lookback parameter, L, that indicates how far back in the sequence the description "looks" in order to predict the next object in the sequence. The above rule has a lookback parameter of 1, because is examines q_{i-1}.

5.3. The Periodic Model

This model consists of rules that describe objects in the sequence as having attribute values that repeat periodically. For example, the rule "Play alternating red and black cards" is a periodic rule. The periodic model has two parameters: the period length, P, and the lookback, L. The period length parameter, P, gives the number of phases in the periodic rule. A periodic rule can be viewed as applying a splitting transformation to split the original sequence into P separate sequences. Each separate phase sequence has a simple description. The lookback parameter, L, tells how far back, within a phase sequence, a periodic rule "looks" in order to predict the attributes of the next object in that phase. The periodic model imposes the additional constraint (or preference) that the different phases be disjoint (i.e., any given card is only playable within one phase).

A periodic rule is represented as an ordered P-tuple of VL22 conjunctions. The j-th conjunct describes the j-th phase sequence. The rule

```
<[color(qi)=red], [rank(qi)≥rank(qi-1)]>
```

is a periodic rule with P=2 and L=1, which says that the sequence is made of two (interleaved) phases. Each card in the first phase is red; each card in the second phase has at least as large a rank as the preceding card in that phase. Hence, one sequence that satisfies this rule is <2H 3C 10H 5S AD 6S 6H 6C>.

A more complex periodic rule is the rule used to generate the sequence shown in Figure 2-1. It can be represented as

```
< [texture-of-nodes(qi)=solid black] &
  [shape(qi)=T-junction] &
  [orientation(qi)=orientation(qi-1)-45],

  [texture-of-nodes(qi)=clear] &
  < [number-of-nodes(qi)=4],
    [number-of-nodes(qi)=8] >,

  [texture-of-nodes(qi)=cross] &
  [shape(qi)=bar] &
  [orientation(qi)=orientation(qi-1)+45] >
```

Notice that this is a periodic rule with three phases and a lookback of 1. The middle phase of the period is itself a periodic rule with the NUMBER-OF-NODES alternating between 4 and 8.

5.4. Derived models

The three basic models can be combined to describe more complex rules. Basic models can be joined by conjunction, disjunction, and negation. For example, the rule "play alternating red and black cards such that the cards are in non-decreasing order" is a conjunction of the periodic rule

```
<  [color(qi)=red], [color(qi)=black]  >
```

and the DNF rule

```
[rank(qi) ≥ rank(qi-1)].
```

5.5. Model Equivalences and the Heuristic Value of Models

The reader may have noticed that the decomposition and periodic models appear to be special cases of the DNF model. For instance, given that the clauses in a decomposition rule are mutually-exclusive and exhaustive, the decomposition rule

$$L_1 \Rightarrow R_1 \,\&$$
$$L_2 \Rightarrow R_2 \,\&$$
$$\vdots$$
$$L_m \Rightarrow R_m$$

can be written as the DNF rule

$$[L_1 \,\&\, R_1] \vee [L_2 \,\&\, R_2] \vee \ldots \vee [L_m \,\&\, R_m]$$

Similarly, if the clauses of a periodic rule are mutually-exclusive and exhaustive, then the periodic rule

$$\langle C_1, C_2, \ldots, C_k \rangle$$

can be expressed as a decomposition rule of the form

$$C_1 \Rightarrow C_2$$
$$C_2 \Rightarrow C_3$$
$$\vdots$$
$$C_{k-1} \Rightarrow C_k$$
$$C_k \Rightarrow C_1.$$

Even when the constraints of mutual exclusion and exhaustion are violated, it is usually possible to develop some equivalent DNF rule for any periodic or decomposition rule. For instance, in the periodic rule

```
< [color(q1-0)=red], [rank(q1-0)=even] >
```
(paraphrase: play alternating red and even cards)

the different phases are overlapping. The above transformation into a decomposition rule

```
[color(q1-1)=red]=>[parity(q1)=even] &
[parity(q1-1)=even]=>[color(q1)=red]
```

does not work, because, for example, the sequence

```
<3D 2D 4C ...>
```

satisfies the second rule (the first if-then clause can be applied twice), but not the first rule (since the 4C is not red). However, it is possible to get around this particular problem by defining a new descriptor for each object in the original sequence, called POSITION, that has the value i for object q_i. With this descriptor, the above rule can be encoded as

```
[position(q1)=odd] => [color(q1)=red]
[position(q1)=even] => [parity(q1)=even]
```

Hence, it appears that all rules can be written as DNF rules.

Given this fact, it is reasonable to ask why multiple models should be used at all. The answer is that the primary value of multiple models is that they provide heuristic guidance to the search for plausible rules. Hence, though the DNF model is capable of *representing* all of these rules, it is not helpful for *discovering* them. In short, it is *epistemologically adequate* but not *heuristically adequate* (see [McCarthy and Hayes, 1969; McCarthy, 1977]). Each model directs the attention of the learning system to a small subspace of the space of all possible DNF VL22 rules. The next section shows how the constraints associated with each model are incorporated into special model-fitting induction algorithms.

6. Architecture and Algorithms

In section 3 we described the three basic processes involved in discovering sequence-generating rules: (a) transformation of the original sequence to obtain a derived sequence, (b) selection of appropriate models for the given sequence, and (c) fitting of the models to the derived sequence. In sections 4 and 5, the four data transformations and the three models were presented. This section covers the third step of fitting the specialized models to the transformed sequence. The model-fitting process is most easily understood in the context of the program architecture, so this section also discusses the architecture in detail.

6.1. Overview of the Program

The processes in the program (see Figure 6-1) are structured into four components—the three basic components mentioned above plus an evaluation step. The processes of transforming the initial sequence and of selecting and parameterizing a model are performed in parallel. Then, specialized model-fitting algorithms use the transformed sequence to instantiate the model to obtain a candidate sequence-generating rule. These candidate rules are then evaluated to determine a final set of rules.

The reason for performing data transformation and model selection in parallel is that these two processes are interdependent. For example, if a periodic model is selected (with period length P), then a splitting transformation (with number of phases P) needs to be applied to the sequence. These two processes can be viewed as simultaneous cooperative searches of two spaces: the space of possible data transformations and the space of possible parameterized models.

6.2. Overview of the Concentric Ring Architecture

In order for the learning program to be easily modified to handle entire classes of NDP problems, the program is structured as a set of concentric knowledge rings (see Figure 6-2). A knowledge ring is a set of routines that perform a particular function using only knowledge appropriate to that function. The procedures within a given ring may invoke other procedures in that ring or in rings that are inside the given ring. Under these constraints, the concentric ring structure forms a hierarchically organized system.

Ideally, the rings should be organized so that the outermost ring uses the most problem-specific knowledge and performs the most problem-specific operations and the inner-most ring uses the most general knowledge and performs the most general tasks. Such an architecture improves the program's generality because it can be applied to increasingly different NDP problems by removing and replacing the outer rings. In order to apply the program to radically different learning problems, all but the inner-most ring may need to be replaced.

The ring architecture is used here as follows. The outer-most rings perform user-interface functions and convert the initial sequence from whatever domain-specific notation is being used into a sequence of

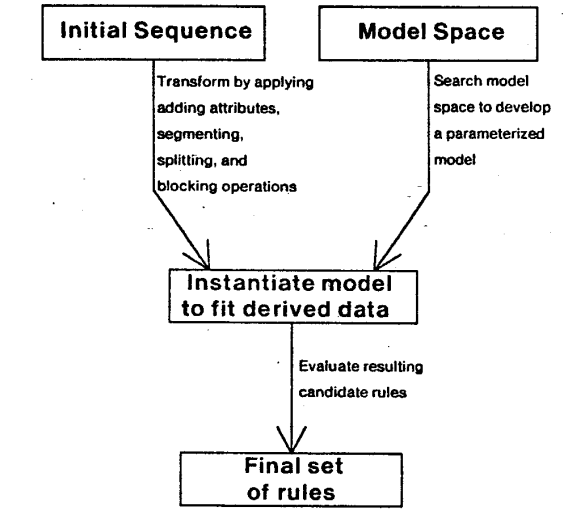

Figure 6-1: The Model-fitting Approach

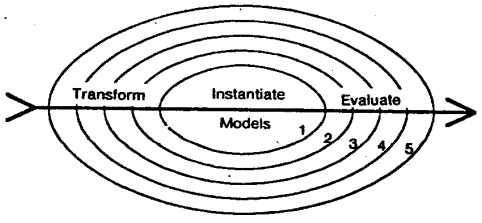

Figure 6-2: The knowledge ring architecture

VL22 events. The inner-most ring performs the model-fitting functions. It expects the data to be properly transformed so that the data have the same form as the models to which they are to be fitted. The intervening rings conduct the simultaneous processes of developing a properly parameterized model and transforming the input sequence into an appropriate form.

The intervening rings also evaluate the rules discovered by the inner-most ring using the knowledge available in each ring.

6.3. The Program SPARC (ELEUSIS version)

SPARC (Sequential PAttern ReCognition) is a general program designed to solve a variety of NDP problems using the ring architecture. So far, we have implemented only a more specific version of the program, called SPARC/E, tailored specifically to the problem of rule discovery in the game Eleusis. SPARC is made up of five rings, as shown in Figure 6-2. This section describes the functions of each ring in the SPARC/E version of the program. To illustrate these ring functions, we use the Eleusis layout shown in Figure 6-3. Recall that in an Eleusis layout, the main line shows the correctly-played sequence of cards (positive examples). The side lines, which branch out below the main line, contain cards that do not satisfy the rule—that is, incorrect continuations of the sequence (negative examples).

```
Main line:  3H  9S  4C  JD  2C  10D  8H  7H  2C
Side lines:     JD      AH  AS          10H
                5D      8H  10S
                        QD
```

Figure 6-3: Sample Eleusis Layout

6.3.1. Ring 5: User Interface

Ring 5, the outer-most ring, provides a user interface to the program. It executes user's commands for playing the card game Eleusis, as well as commands for controlling the search, data transformation, generalization, and evaluation functions of the program. One command in Ring 5 is the INDUCE command that instructs SPARC/E to look for plausible NDP rules that describe the current sequence. When the INDUCE command is given, Ring 5 calls Ring 4 to begin the rule discovery process.

6.3.2. Ring 4: Adding Derived Attributes

Ring 4 applies the adding-derived-attributes transformation to convert the Eleusis layout into a sequence of VL22 events. This involves creating derived attributes that make explicit certain commonly known characteristics of playing cards that are likely to be used in an Eleusis rule: COLOR, PARITY, FACED versus NON-FACED cards, and so on. Figure 6-4 shows the layout from Figure 6-3 after it has been processed by Ring 4. The plusses and minuses along the right-hand side of the figure indicate whether the event is a positive example or a negative example of the sequence-generating rule. These derived events are passed to Ring 3 for further processing.

6.3.3. Ring 3: Segmenting the Layout

Ring 3 is the first Eleusis-independent ring. It applies the segmenting transformation to the sequence supplied by Ring 4. In the present implementation, the end points of each segment are determined by applying a segmentation predicate, $P(card_{i-1}, card_i)$ to all pairs of adjacent events in the sequence. When the predicate P evaluates to FALSE, the sequence is broken between $card_{i-1}$ and $card_i$

```
VL22 event                                   positive or
                                               negative
[rank(card1)=3][suit(card1)=H]
   [parity(card1)=odd][color(card1)=red]
   [prime(card1)=N][faced(card1)=Y]            +
[rank(card2)=9][suit(card2)=S]
   [parity(card2)=odd][color(card2)=black]
   [prime(card2)=N][faced(card2)=N]            +
[rank(card3)=J][suit(card3)=D]
   [parity(card3)=odd][color(card3)=red]
   [prime(card3)=Y][faced(card3)=Y]            -
[rank(card3)=5][suit(card3)=D]
   [parity(card3)=odd][color(card3)=red]
   [prime(card3)=N][faced(card3)=Y]            -
[rank(card3)=4][suit(card3)=C]
   [parity(card3)=even][color(card3)=black]
   [prime(card3)=N][faced(card3)=N]            +
[rank(card4)=J][suit(card4)=D]
   [parity(card4)=odd][color(card4)=red]
   [prime(card4)=Y][faced(card4)=Y]            +
                  etc.
```

Figure 6-4: Derived layout after Ring 4 processing.

to form the end of a segment. Typical segmentation predicates used are:

$$[rank(card_i) = rank(card_{i-1})]$$
$$[rank(card_i) = rank(card_{i-1}) + 1]$$
$$[color(card_i) = color(card_{i-1})]$$
$$[suit(card_i) = suit(card_{i-1})]$$
$$[parity(card_i) = parity(card_{i-1})]$$

Other techniques for performing segmentation, such as providing a predicate that becomes TRUE at a segment boundary (see section 4.2.2), are not implemented in SPARC/E.

Ring 3 searches the space of possible segmentations using two search pruning heuristics. After each attempt to segment the sequence, it counts the number of derived objects (segments), k, in the derived sequence. If k is less than 3, the segmentation is discarded since there are too few derived objects to use for generalization. If k is more than half of the number of objects in the original sequence, the segmentation is also discarded because in this case many segments contain only one original object. Segmented sequences that survive these two pruning heuristics are passed on to Ring 2 for further processing.

One segmentation that Ring 3 always performs is the "null" segmentation—that is, it always passes the unsegmented sequence directly to the inner rings. Figure 6-5 shows a sample layout and the resulting derived layout after segmentation using the segmentation condition: `[suit(card1)=suit(card1+1)]`. The derived objects (segments) are denoted by variables $string_i$. The negative event `[suit(string2)=D][color(string2)=red][length(string2)=3]` is obtained from the segment `<5D 2D 4D>`.

SPARC/E derives the descriptors COLOR, SUIT, and LENGTH to describe each derived object. The choice of which descriptors to derive involves three steps. First, LENGTH is derived whenever the segmentation transformation is applied. Second, any descriptor that is tested in the segmentation predicate (in this case, SUIT) is also derived. Third, any descriptor is derived whose value can be proved to be the same for all cards in each segment. In this case, COLOR is derived because, if SUIT is a constant, then COLOR is also a constant. Using this

```
The layout:

3H  5D  2D  7C  AC  9C  JH  6H  8H  QH  KS
    5S  4D      AH
                7S

The derived sequence:

description of                          positive or
derived object                            negative

[suit(string1)=H][color(string1)=red]
        [length(string1)=1]                   +
[suit(string2)=D][color(string2)=red]
        [length(string2)=2]                   +
[suit(string2)=D][color(string2)=red]
        [length(string2)=3]                   -
[suit(string3)=C][color(string3)=black]
        [length(string3)=3]                   +
[suit(string4)=H][color(string4)=red]
        [length(string4)=4]                   +
```

Figure 6-5: Sample layout and segmented sequence.

segmentation, SPARC can use the DNF model to discover that the segmented sequence can be described as

```
[length(string1)=length(string1-1)+1]
```

That is, the LENGTH of each segment of constant SUIT (in the main line) increases by 1.

6.3.4. Ring 2: Parameterizing the Models

Ring 2 searches the space of parameterizations of the three basic models. Each model is considered in turn. For each model, Ring 2 develops a set of derived events based on each allowed value of the lookback parameter, L, and the number of phases parameter, P. The user can control which models should be inspected and what range of values for L and P should be investigated. By default, the program will inspect the decomposition model with L = 0, 1, or 2, and the periodic model with P = 1 or 2 and L = 0 or 1.

Specifically, Ring 2 performs the following actions depending on which model is being parameterized:

A. For the *decomposition* model with lookback parameter L, Ring 2 applies the blocking transformation to break the sequence received from Ring 3 into blocks of length L. After blocking, all of the attributes that described the original objects are converted into attributes that describe the whole block (as described in section 4 above). Furthermore, sum and difference descriptors are derived to represent the relationships between adjacent objects in the original sequence. The resulting derived events can be viewed as very specific if-then clauses of the following form.

Given an initial sequence of objects $\langle q_1, q_2, ..., q_m \rangle$, let us look at block b_i which describes the subsequence $\langle q_{i-L}, ..., q_{i-1}, q_i \rangle$. Let F_j denote the selectors of object q_{i-j} renamed so that they apply to b_i. For example, F_1 could be the selectors `[suit1(bi)=H][rank1(bi)=3]`—selectors that originally referred to object q_{i-1}. Let $d(F_j, F_k)$ denote all of the difference selectors obtained by "subtracting" event F_k from event F_j, and let $s(F_j, F_k)$ denote all of the summation selectors obtained by "summing" events F_j and F_k. For example, $d(F_0, F_1)$ could include the selectors `[d-suit01(bi)=2][d-rank01(bi)=-3]` obtained from "subtracting" F_1 from F_0.

With these definitions, the derived events for the decomposition model have the form:

$$F_1 \& ... \& F_L \Rightarrow F_0 \& d(F_0, F_1) \& ... \& d(F_0, F_L) \& s(F_0, F_1) \& ... \& s(F_0, F_L)$$

These derived events no longer need to be ordered since the ordering information is made explicit within the events. These events have the form of very specific if-then clauses. This facilitates the model-fitting process in Ring 1..

B. For the *DNF* model with lookback parameter L, the sequence derived in Ring 3 is blocked in a very similar manner, except that only the selectors describing q_i are retained in the description of block b_i. The derived events have the following form:

$$F_0 \& d(F_0, F_1) \& ... \& d(F_0, F_L) \& s(F_0, F_1) \& ... s(F_0, F_L)$$

These events are very specific conjuncts that are passed to the A^q algorithm in Ring 1, where they are generalized to form a DNF description.

C. For the *periodic model* with period length P and lookback L, Ring 2 performs a splitting transformation followed by a blocking transformation. First, the sequence obtained from Ring 3 is split into P separate sequences. Then each separate sequence is blocked into blocks of length L+1. The derived events have the same form as the events derived for the DNF model. Note that because the blocking occurs after the splitting, the lookback takes place only within a phase.

To provide an example of the function of Ring 2, Figure 6-6 shows some events from Figure 6-3 after they have been transformed in preparation for fitting to a decomposition model with L=1.

6.3.5. Ring 1: The basic model-fitting algorithms

Ring 1 consists of three separate model-fitting algorithms: the A^q algorithm, the decomposition algorithm and the periodic algorithm.

The A^q algorithm [Michalski and Kulpa, 1971] is used to fit the DNF model to the data. A^q attempts to find the DNF description with the fewest number of conjunctive terms that covers all of the positive examples and none of the negative examples. The algorithm operates as follows. First, a positive example, called the seed, is chosen, and the set of maximally-general conjunctive expressions consistent with all of the negative examples is computed. This set is

```
[rank1(b2)=3][suit1(b2)=H]
   [parity1(b2)=odd][color1(b2)=red]
   [prime1(b2)=Y][faced1(b2)=N]        =>

   [rank0(b2)=9][suit0(b2)=S][parity0(b2)=odd]
   [color0(b2)=black][prime0(b2)=N]
   [faced0(b2)=N][d-rank01(b2)=+6]
   [d-suit01(b2)=+1][d-parity01(b2)=N]
   [d-color01(b2)=Y][d-prime01(b2)=Y]
   [d-faced01(b2)=Y][s-rank01(b2)=12]          +

[rank1(b3)=9][suit1(b3)=S]
   [parity1(b3)=odd][color1(b3)=black]
   [prime1(b3)=N][faced1(b3)=N]        =>

   [rank0(b3)=J][suit0(b3)=D][parity0(b3)=odd]
   [color0(b3)=red][prime0(b3)=Y]
   [faced0(b3)=Y][d-rank01(b3)=+2]
   [d-suit01(b3)=+2][d-parity01(b3)=N]
   [d-color01(b3)=Y][d-prime01(b3)=Y]
   [d-faced01(b3)=Y][s-rank01(b3)=20]          -
```

Figure 6-6: Some events of Figure 6-3 transformed for decomposition L=1.

called a *star*, and it is equivalent to the G-set in Mitchell's [1978] version space approach. One element from this star is chosen to be a conjunct in the output DNF description, and all positive examples covered by it are removed from further consideration. If any positive examples remain, the process is repeated, selecting as a new seed some positive example that was not covered by *any* member of any preceding star. In this manner, a DNF description with few conjunctive terms is found. If the stars are computed without any pruning, then Λ^9 can provide a tight bound on the number of conjuncts that would appear in the optimal DNF description with fewest conjunctive terms.

The decomposition algorithm is an iterative algorithm that seeks to fit the data to a decomposition model. The key task of the decomposition algorithm is to identify a few attributes, called *decomposition attributes*, from which the decomposition rule can be developed. A *decomposition attribute* is an attribute that appears on the left-hand side of an if-then clause of a decomposition rule. For example, the decomposition rule

```
[color(card1-1)=black] => [parity(card1)=odd] V
[color(card1-1)=red]   => [parity(card1)=even]
```

decomposes on COLOR. Hence, COLOR is the single decomposition attribute.

The algorithm uses a generate-and-test approach of the following form:

```
decompositionattributes := {}    The empty set

while rule is not consistent do
    begin

    generate a trial decomposition
    (based on positive evidence only)
    for each possible decomposition attribute

    test these trial decompositions against
    the data

    select the best decomposition attribute and
    add it to the set decompositionattributes

    end
```

The process of generating a trial decomposition takes place in two steps. First, a VL22 conjunction is formed for each possible value of the decomposition attribute. All positive events that have the same value of the decomposition attribute on their left-hand sides are merged together to form a single conjunction of selectors. This VL22 conjunction forms the right-hand side of a single clause in the decomposition rule. Within this conjunction, a selector is created for each attribute by forming the internal disjunction of the values in the corresponding selectors in the events. For example, using all of the events derived in Ring 2 for the sample layout in Figure 6-3, the decomposition algorithm generates the trial decomposition shown in Figure 6-7 for the PARITY($card_{i-1}$) attribute.

Since there are only two values (ODD and EVEN) for the decomposition attribute in the sequence shown in Figure 6-3, two conjunctions are formed. The first conjunction is obtained by merging all of the positive events for which `[parity(card1-1)=odd]`. There are four such events. The first selector in that conjunction, `[rank(card1)=9 v 4 v 2]`, is obtained by forming the internal disjunction of the values of `rank(card1)` in each of the four events.

The second step in forming a trial decomposition is to generalize each clause in the trial rule. The generalization is accomplished by applying rules of generalization to extend internal disjunctions and drop selectors. (See [Michalski, 1983] for a description of various rules of generalization.) Corresponding attributes in the different clauses of

```
[parity(card1-1)=odd] => [rank(card1)=9 v 4 v 2]
   [suit(card1)=S v C][parity(card1)=even v odd]
   [color(card1)=black][prime(card1)=Y v N]
   [faced(card1)=N]
   [d-rank(card1,card1-1)=+6 v -5 v -7]
   [d-suit(card1,card1-1)=1 v 2 v 3]
   [d-parity(card1,card1-1)=Y v N]
   [d-color(card1,card1-1)=Y v N]
   [d-prime(card1,card1-1)=Y v N]
   [d-faced(card1,card1-1)=Y v N]
   [s-rank(card1,card1-1)=12 v 13 v 9]

[parity(card1-1)=even] =>
   [rank(card1)=J v 10 v 8 v 7]
   [suit(card1)=H v D][parity(card1)=even v odd]
   [color(card1)=red][prime(card1)=Y v N]
   [faced(card1)=Y v N]
   [d-rank(card1,card1-1)=7 v 8 v -2 v -1]
   [d-suit(card1,card1-1)=0 v 1]
   [d-parity(card1,card1-1)=Y v N]
   [d-color(card1,card1-1)=Y v N]
   [d-prime(card1,card1-1)=Y v N]
   [d-faced(card1,card1-1)=Y v N]
   [s-rank(card1,card1-1)=15 v 12 v 18]
```

Figure 6-7: Trial decomposition on the PARITY($card_{i-1}$) attribute

the decomposition rule are compared, and selectors whose value sets overlap are dropped. When these rules of generalization are applied to the trial decomposition for PARITY, for example, the following generalized trial decomposition is obtained:

```
          [parity(card1-1)=odd] =>
    [suit(card1)=C v S][color(card1)=black]
         [parity(card1-1)=even] =>
     [suit(card1)=H v D][color(card1)=red]
```

This is a very promising trial decomposition. However, it has been developed using only positive evidence, and it has been generalized without considering that the generalization may have caused the rule to cover negative events. Hence, the trial decomposition must be tested against the negative events to determine whether or not it is consistent. It turns out that the generalized trial decomposition shown above is indeed consistent with the negative evidence.

After a trial decomposition has been developed for each possible decomposition attribute, the best decomposition attribute is selected according to a heuristic attribute-quality functional. The attribute-quality functional tests such things as the number of negative events covered by the trial decomposition, the number of clauses with non-null right-hand sides, and the complexity of the trial decomposition (defined as the number of selectors that cannot be written with a single operator and a single value). The chosen trial decomposition forms a candidate sequence-prediction rule.

If the candidate rule is not consistent with the data (i.e., still covers some negative examples), then the decomposition algorithm must be repeated to select a second attribute to add to the left-hand sides of the if-then clauses. This has the effect of splitting each of the if-then clauses into several more if-then clauses. For example, if we first decomposed on PARITY($card_{i-1}$) and then on FACED($card_{i-1}$), we would obtain four if-then clauses of the form:

```
[parity(card1-1)=odd][faced(card1-1)=N] => ...
[parity(card1-1)=odd][faced(card1-1)=Y] => ...
[parity(card1-1)=even][faced(card1-1)=N] => ...
[parity(card1-1)=even][faced(card1-1)=Y] => ...
```

The periodic algorithm is nearly the same as the decomposition algorithm. For each phase of the period, it takes all of the positive

events in that phase and combines them to form a single conjunct by forming the internal disjunction all of the value sets of corresponding selectors. Next, rules of generalization are applied to extend internal disjunctions and drop selectors. Finally, corresponding attributes in different phases are compared, and selectors whose values sets overlap are dropped if this can be done without covering any negative examples.

6.3.6. Evaluating the NDP rules

Once Ring 1 has instantiated the parameterized models to produce a set of rules, the rules are passed back through the concentric rings of the program. Each ring evaluates the rules according to plausibility criteria based on knowledge available in that ring. Ring 2, for example, checks to see that the rule does not predict an end to the sequence. It is assumed that a valid sequence can be continued indefinitely. Ring 3 checks the last (partial) segment to see if it is consistent with the rule. It is possible to induce a rule, using only the complete segments, that is not consistent with the final segment. Ring 4 tests the rule using the plausibility criteria for Eleusis. These criteria are:

1. Prefer rules with intermediate degree of complexity. In Eleusis, Occam's Razor does not always apply. The dealer is unlikely to choose a rule that is extremely simple, because it would be too easy to discover. Very complex rules will not be discovered by anyone, and, since the rules of the game discourage such an outcome, the dealer is not likely to choose such complex rules either.

2. Prefer rules with an intermediate degree of non-determinism. Rules with a low degree of non-determinism lead to many incorrect plays, thus rendering them easy to discover. Rules that are very nondeterministic generally lead to few incorrect plays and are therefore difficult to discover.

Rules that do not satisfy these heuristic criteria are discarded. The remaining rules are returned to Ring 5 where they are printed for the user.

7. Examples of Program Execution

In this section, we present some example Eleusis games and the corresponding sequence-generating laws that were discovered by SPARC/E. Each of these games was an actual game among people, and the rules are presented as they were displayed by SPARC/E (with minor typesetting changes).

The raw sequences presented to SPARC/E had only two attributes: SUIT and RANK. SPARC/E was given definitions of the following derivable attributes:

- COLOR (red for Hearts and Diamonds; black for Clubs and Spades)
- FACE (true if card is a faced, picture card, false otherwise)
- PRIME (true if card has a prime rank, false otherwise)
- MOD2 (the parity value of the card, 0 if card is even, 1 otherwise)
- MOD3 (the rank of the card modulo 3)
- LENMOD2 (When SPARC/E segments the main sequence into derived subsequences, it computes the LENGTH of each of the subsequences modulo 2)

Three examples of the program execution are presented: one showing the program at its best, one showing some of the shortcomings of the program, and one demonstrating weaknesses of the program. A few explanations are required. First, each rule is assumed to be universally quantified over all events in the sequence. This quantification is not explicitly printed. Second, when the value set of a selector includes a set of adjacent values (e.g., `[rank(card1)=3 v 4 v 5]`, this is printed as `[RANK(CARDI)=3..5]`. The computation times given are for an implementation in PASCAL on the CDC CYBER 175.

7.1. Example 1

In this example, we show the program discovering a segmented rule. The program was presented with the following layout:

```
Main line:   AH  7C  6C  9S  10H 7H  10D JC  AD
Side lines:          KD          5S      QD
                     JH

continued:   4H  8D  7C  9S  10C KS  2C  10S JS
             3S              9H      QH
                             6H      AD
```

The program only discovered one rule for this layout, precisely the rule that the dealer had in mind (1.2 seconds required):

```
RULE 1: LOOKBACK: 0 NPHASES: 1 PERIODIC MODEL

CRITERION=[COLOR(CARDI)=COLOR(CARDI-1)]:
          PERIOD([LENMOD2(STRINGI)=1])
```

The rule states that one must play strings of cards with the same color. The strings must always have odd length. The CRITERION= gives the segmentation criterion that a segment is a string of cards all of the same color. CARDI refers to the I-th card in the original sequence. STRINGI refers to the I-th string in the segmented sequence. SPARC/E discovered this rule as a degenerate periodic rule with a period length, P, of 1. Actually, the rule that the dealer had in mind had one additional constraint: a queen must not be played adjacent to a jack or king. Rules containing such exception clauses cannot be discovered by SPARC/E.

7.2. Example 2

The second example requires the program to discover a fairly simple periodic rule. SPARC/E discovers three equivalent versions of it.

Here is the layout:

```
Main line:   JC  4D  QH  3S  QD  9H  QC  7H  QD
Side lines:  KC  5S                  4S      10D
                 7S

continued:   9D  QC  3H  KH  4C  KD  6C  JD  8D

continued:   JH  7C  JD  7H  JH  6H  KD
```

The program discovered the following descriptions of this layout (0.49 seconds were required):

```
RULE 1: LOOKBACK: 1 NPHASES: 0
DECOMPOSITION MODEL

[FACE(CARDI-1)=FALSE] =>
   [RANK(CARDI)≥JACK]
   [RANK(CARDI)>RANK(CARDI-1)]
   [FACE(CARDI)=TRUE]            V

[FACE(CARDI-1)=TRUE] =>
   [RANK(CARDI)=3..9]
   [RANK(CARDI)<RANK(CARDI-1)]
   [FACE(CARDI) =FALSE]


RULE 2: LOOKBACK: 1 NPHASES: 1 PERIODIC MODEL

PERIOD([RANK(CARDI)≥3]
       [RANK(CARDI)≠RANK(CARDI-1)]
       [FACE(CARDI)≠FACE(CARDI-1)])


RULE 3: LOOKBACK: 1 NPHASES: 2 PERIODIC MODEL

PERIOD([RANK(CARDI)≥JACK]
       [RANK(CARDI)≥-RANK(CARDI-1)+20]
       [FACE(CARDI)=TRUE],

       [RANK(CARDI)=3..9]
       [RANK(CARDI)=-RANK(CARDI-1)+5..14]
       [FACE(CARDI)=FALSE])
```

Rule 1 is a decomposition rule with a lookback of 1. Rule 2 expresses the rule as a single conjunction. This is possible because FACE versus NON-FACE is a binary condition, and there are precisely two phases to the rule. Rule 3 expresses the rule in the "natural" way as a periodic rule of length 2.

Notice that, although the program has the gist of the rule, it has discovered a number of redundant conditions. For example, in rule 1, the program did not use knowledge of the fact that `[rank(card1)≥jack]` implies `[face(card1)=true]`, and therefore, it did not remove the former selector. Similarly, because of the interaction of the two conditions, `[rank(card1)>rank(card1-1)]` is completely redundant. SPARC/E already has enough background knowledge about the meanings of its attributes to support these inferences. Additional routines need to be written to actually perform them (as is done in the INDUCE-2 program—see [Michalski, 1983]).

7.3. Example 3

The third example shows the upper limits of the program's abilities. During this game, only one of the human players even got close to guessing the rule, yet the program discovers a good approximation of the rule using only a portion of the layout that was available to the human players. Here is the layout:

```
Main line:   4H  6D  8C  JS  2C  6S  AC  6S  10H
Side lines:  7C  6S  KC  AH      6C      AS
             JH  7H  3H  KD
             4C  2C      QS
             10S 7S
             8H  6D
             AD  6H
             2D  4C
```

The program produced the following rules after 6.5 seconds:

```
RULE 1: LOOKBACK: 1 NPHASES: 0 DNF MODEL

[RANK(CARDI)≤5][SUIT(CARDI)=SUIT(CARDI-1)+1] V
[RANK(CARDI)≥6][SUIT(CARDI)=SUIT(CARDI-1)+3]
```

```
RULE 2: LOOKBACK: 1 NPHASES: 1 PERIODIC MODEL

PERIOD([RANK(CARDI)=RANK(CARDI-1)-9]
   [RANK(CARDI)=-RANK(CARDI-1)+4,6,7,11,13,17]
   [SUIT(CARDI)=SUIT(CARDI-1)+1,2,3])


RULE 3: LOOKBACK: 1 NPHASES: 2 PERIODIC MODEL

PERIOD([RANK(CARDI)=ACE,2,8,10]
   [RANK(CARDI)=-RANK(CARDI-1)+1,8,9,10],

   [RANK(CARDI)=5..JACK][SUIT(CARDI)=SPADES]
   [RANK(CARDI)=RANK(CARDI-1)+-0..6]
   [RANK(CARDI)=-RANK(CARDI-1)+8..14]
   [SUIT(CARDI)=SUIT(CARDI-1)+0..2]
   [COLOR(CARDI)=BLACK][PRIME(CARDI)=PTRUE]
   [PRIME(CARDI)=PRIME(CARDI-1)]
   [MOD2(CARDI)=1][MOD2(CARDI)=MOD2(CARDI-1)+0]
   [MOD2(CARDI)=-MOD2(CARDI-1)+0][MOD3(CARDI)=2]
   [MOD3(CARDI)=MOD3(CARDI-1)+0]
   [MOD3(CARDI)=-MOD3(CARDI-1)+1])
```

The rule that the dealer had in mind was:

```
[SUIT(CARDI)=SUIT(CARDI-1)+1]
[RANK(CARDI)≥RANK(CARDI-1)] V

[SUIT(CARDI)=SUIT(CARDI-1)+3]
[RANK(CARDI)≤RANK(CARDI-1)]
```

There is a strong symmetry in this rule: the players may either play a higher card in the next "higher" suit (recall that the suits are cyclically ordered) or a lower card in the next "lower" suit. The program discovered a slightly simpler version of the rule (rule 1) that happened to be consistent with the training instances. Note that adding 3 to the SUIT has the effect of computing the next lower suit.

The other two rules discovered by the program are very poor. They are typical of the kinds of rules that the program discovers when the model does not fit the data very well. Both rules are filled with irrelevant descriptors and values. The current program has very little ability to assess how well a model fits the data. These rules should not be printed by the program since they are highly implausible.

8. Summary

We have presented here a methodology for discovering sequence-generating rules for the nondeterministic prediction problem. The main ideas behind this methodology are

1. the use of task-oriented transformations of the initial data and

2. the use of different rule models to guide the search for sequence-generating rules.

Four different task-oriented transformations (adding attributes, blocking, splitting into phases, and segmenting) and three models (DNF, periodic, and decomposition) have been presented.

The main part of the methodology has been implemented in the program SPARC/E and applied to the NDP problem that arises in the card game Eleusis. The performance of the program indicates that it can discover quite complex and interesting rules.

This methodology is quite general and can be applied to other nondeterministic prediction problems in which the objects in the initial sequence are describable by a small set of finite-valued attributes. The main strengths of the method are (a) that it can solve

learning problems in which the initial training instances require substantial task-oriented transformation and (b) that it can search very large spaces of possible rules using a set of rule models for guidance.

Many aspects of this methodology remain to be investigated. We have not considered NDP problems in which (a) the training instances are noisy, (b) the training instances have internal structure so that an attribute vector representation cannot be used, and (c) the sequence-generating rules are permitted to have exceptions. Application of this methodology to real world problems will probably also require the development of additional sequence transformations and rule models. Also, more heuristics need to be developed that can be used to guide the application of transformations and models.

The implementation of the methodology in program SPARC/E has demonstrated that the method can be used to discover many Eleusis secret rules. There are some shortcomings of the implementation, however. The program presently conducts a nearly exhaustive depth-first search of the possible models and transformations. Much could be gained by having the program conduct a best-first heuristically-guided search instead. Another weakness of SPARC/E is its poor ability to evaluate the plausibility of the rules it discovers. It is also not able to simplify rules by removing redundant selectors, nor is it able to estimate the degree of nondeterminism of the rule. Both of these can be implemented without too much difficulty by including inference routines that make more complete use of the background knowledge already available to the program. Finally, an important weakness of the program is its inability to form composite models. SPARC/E is not presently able to handle the NDP problem shown in Figure 1, because it involves a periodic rule in which one of the phases contains an imbedded periodic sequence (see section 5.3).

In addition to these specific problems, there are some more general problems that further research in the area of sequence-generating laws should address. First, in some real world problems, there are several example sequences available for which the sequence-generating law is believed to be the same. Such problems occur, in particular, in describing the process of disease development in medicine and agriculture. A specific problem of this type that has been partially investigated involves predicting the time course of cutworm infestation in a cornfield and estimating the potential damage to the crop (see [Davis, 1981], [Baim, 1983], and [Boulanger, 1983]). In this problem, several sequences of observations are available—one for each field—and there is a need to develop a sequence-generating law that predicts all of these sequences.

A second general problem for further research is to handle continuous processes. AI research has so far given little attention to this case.

References

Abbott, R., "The New Eleusis," Available from the author, Box 1175, General Post Office, New York, NY 10116, 1977.

Amarel, S., "On Representations of Problems of Reasoning About Actions," *Machine Intelligence 3*, Michie, D., (ed.), University of Edinburgh Press, Edinburgh, pp. 131-171, 1968.

Baim, P. W., "Automated Acquisition of Decision Rules: Problems of Attribute Construction and Selection," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1983.

Boulanger, A. G., "The Expert System PLANT/CD: A Case Study in Applying the General Purpose Inference System ADVISE to Predicting Black Cutworm Damage in Corn," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1983.

Buchanan, B. G., and Mitchell, T. M., "Model-Directed Learning of Production Rules," in *Pattern-directed Inference Systems*, Waterman, D. A., and Hayes-Roth, F., (eds.), Academic Press, New York, 1978.

Cohen, P., and Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, Vol. III, Kaufmann, Los Altos, 1982.

Davis, J., "CONVART: A Program for Constructive Induction on Time-Dependent Data," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1981.

Dietterich, T. G., London, R., Clarkson, K, and Dromey, G., "Learning and Inductive Inference," Chapter XIV in Vol. 3 of *The Handbook of Artificial Intelligence*, Cohen, P. R., and Feigenbaum, E. A., (eds.). 1982.

Dietterich, T. G., and Michalski, R. S., "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods," *Artificial Intelligence*, 1981.

Engelmore, R., and Terry, A., "Structure and Function of the Crysalis System," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1979.

Friedland, P. E., "Knowledge-Based Experiment Design in Molecular Genetics," Rep. No. HPP-79-29, Department of Computer Science, Stanford University, 1979.

Gardner, M., "On Playing the New Eleusis, the game that simulates the search for truth," *Scientific American*, No. 237, pp. 18-25, October, 1977.

Hedrick, C. L., "Learning Production Systems from Examples," *Artificial Intelligence*, Vol. 7, No. 1, pp. 21-49, 1976.

Karpinski, J., and Michalski, R. S., "A System that Learns to Recognize Hand-written Alphanumeric Characters," *Proce Institute Automatyki, Polish Academy of Sciences*, No. 35, 1966.

Kotovsky, K., and Simon, H. A., "Empirical Tests of a Theory of Human Acquisition of Concepts for Sequential Patterns," *Cognitive Psychology*, No. 4, pp. 399-424, 1973.

Langley, P. W., "Descriptive Discovery Processes: Experiments in Baconian Science," Rep. No. CMU-CS-80-121, Department of Computer Science, Carnegie-Mellon University, 1980.

McCarthy, J., "Programs with Common Sense," in *Proceedings of the Symposium on the Mechanization of Thought Processes*, National Physical Laboratory, pp. 77-84, 1958.

McCarthy, J., and Hayes, P., "Some Epistemological Problems from the Standpoint of Artificial Intelligence," in *Machine Intelligence 4*, Meltzer, B., and Michie, D., (eds.), Edinburgh University Press, Edinburgh, pp. 463-502, 1969.

Michalski, R. S., "A Theory and Methodology of Inductive Learning," *Artificial Intelligence*, Vol. 20, 111-161, 1983.

Michalski, R. S., and Chilausky, R. L., "Learning by Being Told and Learning From Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems*, Vol. 4, No. 2, June 1980.

Michalski, R. S., and Kulpa, Z., "A System of Programs for the Synthesis of Switching Circuits Using the Method of Disjoint Stars," *Information Processing 71*, pp. 61-65, North-Holland, 1971.

Mitchell, T. M., "Version Spaces: An approach to concept learning," Rep. No. STAN-CS-78-711, December, 1978.

Mitchell, T. M., Utgoff, P. E., and Banerji, R. B., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," in *Machine Learning*, Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., (eds.), Tioga, Palo Alto, 1983.

Persson, S., "Some Sequence Extrapolating Programs: A Study of Representation and Modeling in Inquiring Systems," Rep. No. CS50, 1966.

Samuel, A. L., "Some Studies in Machine Learning using the Game of Checkers," in *Computers and Thought*, Feigenbaum, E. A., and Feldman, J., (eds.) McGraw-Hill, New York, pp. 71-105, 1963.

Samuel, A. L., "Some Studies in Machine Learning using the Game of Checkers II—Recent Progress," *IBM Journal of Research and Development*, Vol. 11, No. 6, pp. 601-617, 1967.

Schank, R., and Abelson, R., "Scripts, Plans, and Knowledge," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 151-157, 1975.

Simon, H. A., "Complexity and the Representation of Patterned Sequences of Symbols," *Psych. Review*, Vol. 79, No. 5, 369-382, 1972.

Simon, H. A., and Kotovsky, K., "Human Acquisition of Concepts for Sequential Patterns," *Psychological Review*, Vol. 70, pp. 534-546, 1963.

Solomonoff, R. S., "A Formal Theory of Inductive Inference," *Information and Control*, Vol. 7, pp. 1-22, 224-254, 1964.

Soloway, E. M., "Learning = Interpretation + Generalization: A Case Study in Knowledge-Directed Learning," Rep. No. COINS TR-78-13, Computer and Information Science Dept., U. Mass. at Amherst, 1978.

Winston, P. H., "Learning Structural Descriptions from Examples," Rep. No. AI-TR-231, MIT, 1970.

APPENDIX

Notational conventions

The following notational conventions are employed in this paper. In general, lowercase letters denote objects in some sequence (q, ph, b) or index variables (i, j, k) or the lengths of sequences (m, n). Uppercase letters denote sets of objects, attributes, and so on (Q, F, S) as well as parameters of models and transformations (L, P). Small capitals denote attributes (COLOR, RANK) and their values (RED, KING).

<>	Angle brackets denote sequences of objects, e.g., <2 4 6 8> and also periodic rules, e.g., <[color(qi)=red],[color(qi)=black]>.
q_i or **qi**	The i-th object in an input sequence.
q_i'	The i-th object in a derived sequence.
q_i^-	An object that constitutes an incorrect extension of the sequence after object q_i.
b_i or **bi**	The i-th block in a sequence derived by the blocking transformation.
ph_i	The i-th phase derived by the splitting transformation.
F	The starting set of attributes for a transformation.
S	The starting set of sequences for a transformation.
Q	The starting set of objects for a transformation.
F'	The set of derived attributes from a transformation.
S'	The set of derived sequences from a transformation.
Q'	The set of derived objects from a transformation.
F_j	The set of selectors describing object q_{i-j} in block b_i.
g	The sequence-generating function that maps a sequence into a set of objects Q_{k+1} that can appear as continuations of the sequence.
Q_{k+1}	The set of objects that can appear as continuations of the sequence $\langle q_1, q_2, ..., q_k \rangle$.
P	The number of phases parameter of the splitting transformation and the periodic model.
L	The lookback parameter of the blocking transformation and all three models.
$[f_i(q_j)=r_k]$ or **[fi(qj)=rk]**	A simple selector, which asserts that feature f_i of object q_j has the value r_k.
[fi(qj)=r1 v r2 v r3]	A selector containing an internal disjunction. It asserts that f_i can have the value r_1 or r_2 or r_3.
d prefix	The **d** prefix on an attribute name indicates that it is a difference attribute. Hence, D-RANK(q_i,q_{i-1}) is equal to RANK(q_i) − RANK(q_{i-1}).
s prefix	The **s** prefix on an attribute name indicates that it is a summation attribute. Hence, S-RANK(q_i,q_{i-1}) is equal to RANK(q_i) + RANK(q_{i-1}).
$d(F_i, F_j)$	The set of difference selectors obtained by "subtracting" selectors F_j from F_i.
$s(F_i, F_j)$	The set of summation selectors obtained by u"adding" selectors F_i and F_j.
=>	Logical implication.