Machine Learning, 5, 239–266 (1990)

Learning Logical Definitions from Relations

J.R. QUINLAN
Basser Department of Computer Science, University of Sydney, Sydney NSW Australia 2006

Editor: Jack Mostow

Abstract. This paper describes FOIL, a system that learns Horn clauses from data expressed as relations. FOIL is based on ideas that have proved effective in attribute-value learning systems, but extends them to a first-order formalism. This new system has been applied successfully to several tasks taken from the machine learning literature.

Keywords. Induction, first-order rules, relational data, empirical learning

1. Introduction

Concept learning, which Hunt, Marin, and Stone (1966) describe succinctly as "[the] capacity to develop classification rules from experience," has long been a principal area of machine learning research. Supervised concept learning systems are supplied with information about several entities whose class membership is known and produce from this a characterization of each class.

One major dimension along which to differentiate concept learning systems is the complexity of the input and output languages that they employ. At one extreme are learning systems that use a *propositional* attribute-value language for describing entities and classification rules. The simplicity of this formalism allows such systems to deal with large volumes of data and thus to exploit statistical properties of collections of examples and counter-examples of a concept. At the other end of the spectrum, *logical inference* systems accept descriptions of complex, structured entities and generate classification rules expressed in first-order logic. These typically have access to background knowledge pertinent to the domain and so require fewer entity descriptions. FOIL, the system described in this paper, builds on ideas from both groups. Objects are described using relations and from these FOIL generates classification rules expressed in a restricted form of first-order logic, using methods adapted from those that have evolved in attribute-value learning systems.

The following section reviews briefly two fundamental methods for learning concepts from attribute-value descriptions of objects. After illustrating inadequacies of this propositional description language, the paper introduces the more powerful first-order formalism used by FOIL. The algorithm itself is described in Section 4. The next section presents results obtained by FOIL on six diverse tasks taken from the machine learning literature. Section 6 analyzes limitations of the current algorithm along with plans for overcoming some of them. The final section discusses FOIL in relation to other methods of learning logical descriptions.

2. Propositional methods for inductive learning

There has been considerable research on the learning formalism in which *objects*, described in terms of a fixed collection of *attributes*, belong to one of a small number of mutually exclusive and exhaustive *classes*. The learning task may be stated as: given a training set of objects whose classes are known, find a rule for predicting the class of an unseen object as a function of its attribute values. This section reviews two approaches to this task and then turns to shortcomings of this representation.

2.1. The divide-and-conquer method

Several systems based on this attribute-value representation express what is learned as a *decision tree*, examples being Hunt et al.'s (1966) CLS, Quinlan's (1979, 1986) ID3, Breiman, Friedman, Olshen and Stone's (1984) CART, and Cestnik, Kononenko and Bratko's (1987) ASSISTANT. At a somewhat oversimplified level, these systems use the same method to construct a decision tree from a training set of objects, summarized as follows:

- If all training objects belong to a single class, the tree is a leaf labelled with that class.
- Otherwise,
  - select a test based on one attribute,
  - divide the training set into subsets, each corresponding to one of the possible (mutually exclusive) outcomes of the test, and
  - apply the same procedure to each subset

The extent to which this method yields compact trees with high predictive accuracy on unseen objects depends on the choice of a test to divide the training set. A good choice should assist the development of pure subsets which do not require further division. It has proved possible to design simple information-based or encoding-length heuristics that provide effective guidance for division, to the extent that this greedy algorithm, which never reconsiders the choice of a test, is adequate for most tasks. Examples of such heuristics appear in Quinlan (1988) and Quinlan and Rivest (1989).

2.2. The covering method

Other induction algorithms, notably the AQ family (Michalski, 1989; Michalski, Mozetič, Hong, and Lavrač, 1986), represent classification knowledge as a disjunctive logical expression defining each class. The core of this "covering" method can be summarized as:

- Find a conjunction of conditions that is satisfied by some objects in the target class, but no objects from another class;
- Append this conjunction as one disjunct of the logical expression being developed;
- Remove all objects that satisfy this conjunction and, if there are still some remaining objects of the target class, repeat the procedure.

Members of the AQ family find a suitable conjunction starting from a single seed object of the target class. Specifically, they carry out a beam search on the space of subsets of the primitive descriptors of the seed objects, looking for simple conjunctions that delineate many objects in the target class. A later algorithm, CN2 (Clark and Niblett, 1987, 1989), shares this core but uses a divide-and-separate method similar to that employed by the decision tree systems to construct a suitable conjunction. Rivest (1988) shows that disjunctive covers in which the complexity of individual conjunctions is bounded are polynomially learnable.

3. Moving to a more powerful representation

The learning methods above have proved their worth in a great variety of applications. Their principal weakness derives from a lack of expressive power in the languages used to describe objects and classification rules. An object must be specified by its values for a fixed set of attributes, and rules must be expressed as functions of these same attributes.

To see why this presents a problem, consider a domain involving directional networks like the one in Figure 1, and suppose that we attempted to set down a collection of attributes sufficient to describe any network. Now, this description can be viewed as an encoding task. A fixed number of attribute values conveys a fixed number of bits of information and it is easy to construct a network that would require more bits to represent its nodes and links. Further, suppose the description task were simplified by restricting networks to a maximum of ten nodes, say, with each node connected to at most three others. Any such network could then be represented in terms of the thirty attributes

attributes A_1, B_1, C_1: the nodes to which node 1 is linked
attributes A_2, B_2, C_2: the nodes to which node 2 is linked

and so on, perhaps using a value of zero to denote "not applicable." Even then, learned concepts would have to be expressed as logical functions of these attribute values. Consider the concept, "two nodes are linked to each other." The expression of this concept in the above representation is the truly horrific

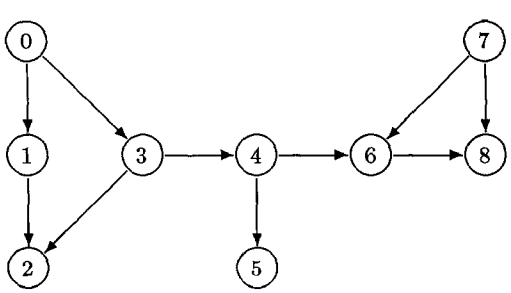

Figure 1. A small network illustrating the limitations of propositional descriptions.

$$
\begin{aligned}
&(A_1 = 2 \vee B_1 = 2 \vee C_1 = 2) \;\&\; (A_2 = 1 \vee B_2 = 1 \vee C_2 = 1)\\
\vee\ &(A_1 = 3 \vee B_1 = 3 \vee C_1 = 3) \;\&\; (A_3 = 1 \vee B_3 = 1 \vee C_3 = 1)\\
\vee\ &(A_1 = 4 \vee B_1 = 4 \vee C_1 = 4) \;\&\; (A_4 = 1 \vee B_4 = 1 \vee C_4 = 1)\\
&\ldots\\
\vee\ &(A_2 = 3 \vee B_2 = 3 \vee C_2 = 3) \;\&\; (A_3 = 2 \vee B_3 = 2 \vee C_3 = 2)\\
\vee\ &(A_2 = 4 \vee B_2 = 4 \vee C_2 = 4) \;\&\; (A_4 = 2 \vee B_4 = 2 \vee C_4 = 2)\\
&\ldots
\end{aligned}
$$

and so on. It is clear that such a propositional description language flounders when faced with complex objects and concepts.

Structural information such as the network can be represented naturally as a collection of *relations*. A relation is associated with a k-ary predicate and consists of the set of k-tuples of constants that satisfy that predicate. In the above network, for example, the predicate *linked-to*(X, Y), denoting that node X is directly linked to node Y, can be represented by the relation

$$
\begin{aligned}
\textit{linked-to} = \{&\langle 0,1\rangle, \langle 0,3\rangle, \langle 1,2\rangle, \langle 3,2\rangle, \langle 3,4\rangle,\\
&\langle 4,5\rangle, \langle 4,6\rangle, \langle 6,8\rangle, \langle 7,6\rangle, \langle 7,8\rangle\}
\end{aligned}
$$

Now we need a language to express what is learned from such relations. One candidate is function-free Horn clause logic, a subset of pure Prolog, in which the clause

$$C \leftarrow L_1, L_2, \ldots, L_n$$

is interpreted as "if L_1 and L_2 and ... and L_n then C." In this paper we adopt an extension in which a clause may contain negated literals on its right-hand side; thus C is a predicate and each L_i is either a predicate or the negation of a predicate. We follow the usual Prolog convention of starting variables with a capital letter, all other atoms being constants. To continue the illustration, the concept "two nodes are linked to each other" might now be expressed as

$$\textit{linked-to-each-other}(X, Y) \leftarrow \textit{linked-to}(X, Y), \textit{linked-to}(Y, X).$$

The propositional attribute-value formalism is simple but limited. Having specified a more powerful language to describe objects and concepts, the next step is to develop learning methods capable of exploiting it.

4. FOIL, a system that constructs definitions

This section introduces a learning algorithm called FOIL whose ancestry can be traced to both AQ and ID3. The task addressed by this system is to find, for one *target* relation at a time, clauses as above that define the relation in terms of itself and the other relations.[1] FOIL thus moves from an explicit representation of the target relation (as a set of tuples of a particular collection of constants), to a more general, functional definition that might

be applied to different constants. For example, an explicit representation of the relation *can-reach* in the network of Figure 1 is

$$\begin{aligned} \textit{can-reach} = \{&\langle 0,1\rangle, \langle 0,2\rangle, \langle 0,3\rangle, \langle 0,4\rangle, \langle 0,5\rangle, \langle 0,6\rangle, \langle 0,8\rangle,\\ &\langle 1,2\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 3,5\rangle, \langle 3,6\rangle, \langle 3,8\rangle, \langle 4,5\rangle,\\ &\langle 4,6\rangle, \langle 4,8\rangle, \langle 6,8\rangle, \langle 7,6\rangle, \langle 7,8\rangle\} \end{aligned}$$

This extensional definition of *can-reach* is applicable only to the given network but, from this relation and the relation *linked-to* given previously, FOIL constructs the general definition

$$\begin{aligned} &\textit{can-reach}(X_1, X_2) \leftarrow \textit{linked-to}(X_1, X_2)\\ &\textit{can-reach}(X_1, X_2) \leftarrow \textit{linked-to}(X_1, X_3), \textit{can-reach}(X_3, X_2) \end{aligned}$$

which is valid for any network.

For a particular target relation, FOIL finds clauses one at a time, removing tuples explained by the current clause before looking for the next in the manner of AQ and CN2. Like ID3, FOIL uses an information-based heuristic to guide its search for simple, general clauses.

4.1. Description of the approach

Consider a target relation based on a k-ary predicate $P(X_1, X_2, \ldots, X_k)$. In place of a training set of objects we have a set of k-tuples of constants, each tuple representing a value assignment to the variables $X_1, X_2, \ldots, X_k$. By analogy with an object's class, each of these tuples is labelled with a $\oplus$ or $\ominus$ to indicate whether or not it is in the target relation. The $\oplus$ tuples are just those given for the target relation *P*. Possible sources of $\ominus$ tuples are:

- As part of the problem, we might be given an explicit list of tuples not in the relation. For instance, we might be told that $\langle 0,2\rangle$ is not in the relation *linked-to*.
- The more usual case would correspond to the closed-world assumption of Prolog (and databases): if a tuple is not explicitly given as a component of the relation, then it is not in the relation. In this case the $\ominus$ tuples consist of all constant k-tuples other than those marked $\oplus$.
- Some domains may have associated *types* that constrain the tuples in a relation. For example, a tuple in the relation *lives-in*(*X*, *Y*) might only make sense if the first constant identified a person and the second a city. In such cases 'all constant k-tuples' above could be amended to 'all constant k-tuples obeying the type constraints of the target relation.'

At the outermost level, the operation of FOIL can be summarized as:

- Establish the training set consisting of constant tuples, some labelled $\oplus$ and some $\ominus$.
- Until there are no $\oplus$ tuples left in the training set:
  - Find a clause that characterizes part of the the target relation.
  - Remove all tuples that satisfy the right-hand side of this clause from the training set.

In the inner loop, FOIL seeks a Prolog clause of the form

$$P(X_1, X_2, \ldots, X_k) \leftarrow L_1, L_2, \ldots, L_n$$

that characterizes some subset of the relation P. The clause is 'grown' by starting with just the left-hand side and adding literals one by one to the right-hand side. At any stage in its development, the partial clause will contain m distinct *bound* variables that appear in the left-hand side and unnegated literals of the right-hand side. This inner loop makes use of a local training set consisting of labelled m-tuples of constants; each tuple in this set represents a value assignment to all bound variables in the clause. The inner loop can be sketched as follows:

- Initialize the local training set T_1 to the training set and let $i = 1$.
- While T_i contains $\ominus$ tuples:
  - Find a literal L_i to add to the right-hand side of the clause.
  - Produce a new training set T_{i+1} based on those tuples in T_i that satisfy L_i. If L_i introduces new variables, each such tuple from T_i may give rise to several (expanded) tuples in T_{i+1}. The label of each tuple in T_{i+1} is the same as that of the parent tuple in T_i.
  - Increment i and continue.

Consider the task of finding the first literal L_1 in a clause. At this stage we have a training set T_1 containing k-tuples, T_1^+ of them labelled $\oplus$ and T_1^- labelled $\ominus$. It may help to bear in mind that each k-tuple corresponds to a possible binding of the variables $\{X_1, X_2, \ldots, X_k\}$ in the left-hand side of the clause.

A good choice for L_1 would be a literal that is satisfied by many of the $\oplus$ tuples in T_1 but few of the $\ominus$ ones, so that testing L_1 would tend to concentrate the $\oplus$ tuples. Let Q be some r-ary predicate and consider choosing as L_1 the literal

$$Q(V_1, V_2, \ldots V_r), \text{ where } V_i \in \{X_1, X_2, \ldots, X_k\} \cup \{Y_1, Y_2, \ldots, Y_s\}$$

(the Y_i's being new variables introduced by this literal). A constant k-tuple $\langle c_1, c_2, \ldots, c_k \rangle$ in the set T_1 satisfies L_1 if, and only if, after binding X_1 to c_1, X_2 to c_2, $\ldots$, X_k to c_k, there exists one or more bindings of the new variables $\{Y_1, Y_2, \ldots, Y_s\}$ such that $\langle V_1, V_2, \ldots, V_r \rangle$ is in the relation Q. Let T_1^{++} denote the number of such tuples that satisfy L_1.

If this literal were chosen for L_1, we would have a new training set T_2 consisting of (k + s)-tuples, T_2^+ of them being labelled $\oplus$ and T_2^- of them $\ominus$. The desirability of this outcome can be estimated by a simple information-based heuristic function of T_1^{++}, T_2^+, and T_2^- as described below in Section 4.3. If T_2 contains only $\oplus$ tuples, this clause is complete, since it defines a subset of the relation P; otherwise, we continue searching for L_2 using the local training set T_2. Note that this is a covering algorithm similar to AQ rather than a divide-and-conquer algorithm like ID3; the addition of each literal to the right-hand side of the clause modifies the existing training set rather than splitting it into separate subsets.

4.2. An example

To illustrate the above, we will trace FOIL's operation on the target relation *can-reach* from the network in Figure 1. There are nine constants so, under the closed-world assumption, the training set T_1 for the first clause

$$can\text{-}reach(X_1, X_2) \leftarrow \ldots$$

consists of the 81 labelled pairs

$\oplus$: ⟨0,1⟩ ⟨0,2⟩ ⟨0,3⟩ ⟨0,4⟩ ⟨0,5⟩ ⟨0,6⟩ ⟨0,8⟩ ⟨1,2⟩ ⟨3,2⟩ ⟨3,4⟩
⟨3,5⟩ ⟨3,6⟩ ⟨3,8⟩ ⟨4,5⟩ ⟨4,6⟩ ⟨4,8⟩ ⟨6,8⟩ ⟨7,6⟩ ⟨7,8⟩
$\ominus$: ⟨0,0⟩ ⟨0,7⟩ ⟨1,0⟩ ⟨1,1⟩ ⟨1,3⟩ ⟨1,4⟩ ⟨1,5⟩ ⟨1,6⟩ ⟨1,7⟩ ⟨1,8⟩
⟨2,0⟩ ⟨2,1⟩ ⟨2,2⟩ ⟨2,3⟩ ⟨2,4⟩ ⟨2,5⟩ ⟨2,6⟩ ⟨2,7⟩ ⟨2,8⟩ ⟨3,0⟩
⟨3,1⟩ ⟨3,3⟩ ⟨3,7⟩ ⟨4,0⟩ ⟨4,1⟩ ⟨4,2⟩ ⟨4,3⟩ ⟨4,4⟩ ⟨4,7⟩ ⟨5,0⟩
⟨5,1⟩ ⟨5,2⟩ ⟨5,3⟩ ⟨5,4⟩ ⟨5,5⟩ ⟨5,6⟩ ⟨5,7⟩ ⟨5,8⟩ ⟨6,0⟩ ⟨6,1⟩
⟨6,2⟩ ⟨6,3⟩ ⟨6,4⟩ ⟨6,5⟩ ⟨6,6⟩ ⟨6,7⟩ ⟨7,0⟩ ⟨7,1⟩ ⟨7,2⟩ ⟨7,3⟩
⟨7,4⟩ ⟨7,5⟩ ⟨7,7⟩ ⟨8,0⟩ ⟨8,1⟩ ⟨8,2⟩ ⟨8,3⟩ ⟨8,4⟩ ⟨8,5⟩ ⟨8,6⟩
⟨8,7⟩ ⟨8,8⟩

If the first literal selected for the right-hand side were *linked-to*(X_1, X_2), ten of these tuples would be satisfied (namely those in the relation *linked-to*). Since all of them are $\oplus$ tuples, this would complete the first clause

$$can\text{-}reach(X_1, X_2) \leftarrow linked\text{-}to(X_1, X_2).$$

On the second time through, we enter the inner loop with T_1 consisting of the remaining $\oplus$ tuples

$\oplus$: ⟨0,2⟩ ⟨0,4⟩ ⟨0,5⟩ ⟨0,6⟩ ⟨0,8⟩ ⟨3,5⟩ ⟨3,6⟩ ⟨3,8⟩ ⟨4,8⟩

and the same $\ominus$ tuples as before. If we now selected the literal *linked-to*(X_1, X_3), the $\ominus$ pairs ⟨2,...⟩, ⟨5,...⟩ and ⟨8,...⟩ would be eliminated. Since this literal introduces a new variable X_3, each remaining pair $\langle X,Y\rangle$ would give triples $\{\langle X,Y,Z_i\rangle\}$, one for each pair $\langle X,Z_i\rangle$ in the relation *linked-to*. The set T_2 would thus consist of the triples

$\oplus$: ⟨0,2,1⟩ ⟨0,2,3⟩ ⟨0,4,1⟩ ⟨0,4,3⟩ ⟨0,5,1⟩ ⟨0,5,3⟩ ⟨0,6,1⟩
⟨0,6,3⟩ ⟨0,8,1⟩ ⟨0,8,3⟩ ⟨3,5,2⟩ ⟨3,5,4⟩ ⟨3,6,2⟩ ⟨3,6,4⟩
⟨3,8,2⟩ ⟨3,8,4⟩ ⟨4,8,5⟩ ⟨4,8,6⟩
$\ominus$: ⟨0,0,1⟩ ⟨0,0,3⟩ ⟨0,7,1⟩ ⟨0,7,3⟩ ⟨1,0,2⟩ ⟨1,1,2⟩ ⟨1,3,2⟩
⟨1,4,2⟩ ⟨1,5,2⟩ ⟨1,6,2⟩ ⟨1,7,2⟩ ⟨1,8,2⟩ ⟨3,0,2⟩ ⟨3,0,4⟩
⟨3,1,2⟩ ⟨3,1,4⟩ ⟨3,3,2⟩ ⟨3,3,4⟩ ⟨3,7,2⟩ ⟨3,7,4⟩ ⟨4,0,5⟩
⟨4,0,6⟩ ⟨4,1,5⟩ ⟨4,1,6⟩ ⟨4,2,5⟩ ⟨4,2,6⟩ ⟨4,3,5⟩ ⟨4,3,6⟩
⟨4,4,5⟩ ⟨4,4,6⟩ ⟨4,7,5⟩ ⟨4,7,6⟩ ⟨6,0,8⟩ ⟨6,1,8⟩ ⟨6,2,8⟩
⟨6,3,8⟩ ⟨6,4,8⟩ ⟨6,5,8⟩ ⟨6,6,8⟩ ⟨6,7,8⟩ ⟨7,0,6⟩ ⟨7,0,8⟩
⟨7,1,6⟩ ⟨7,1,8⟩ ⟨7,2,6⟩ ⟨7,2,8⟩ ⟨7,3,6⟩ ⟨7,3,8⟩ ⟨7,4,6⟩
⟨7,4,8⟩ ⟨7,5,6⟩ ⟨7,5,8⟩ ⟨7,7,6⟩ ⟨7,7,8⟩

Since there are still ⊖ tuples, this clause is not complete; selecting a second literal *can-reach*(X_3, X_2) determines T_3 as

⊕: ⟨0,2,1⟩ ⟨0,2,3⟩ ⟨0,4,3⟩ ⟨0,5,3⟩ ⟨0,6,3⟩ ⟨0,8,3⟩ ⟨3,5,4⟩
⟨3,6,4⟩ ⟨3,8,4⟩ ⟨4,8,6⟩

and, since T_3 contains only ⊕ tuples, the second clause is finalized as the recursive

can-reach(X_1, X_2) ← *linked-to*(X_1, X_3), *can-reach*(X_3, X_2).

All ⊕ tuples in the original relation are covered by one or other of these clauses so the definition of *can-reach* is now complete.

Instead of the second literal above, we could have chosen the literal *linked-to*(X_3, X_2) which would also give a T_3 with only ⊕ tuples

⊕: ⟨0,2,1⟩ ⟨0,2,3⟩ ⟨0,4,3⟩ ⟨3,5,4⟩ ⟨3,6,4⟩ ⟨4,8,6⟩

leading to the clause

can-reach(X_1, X_2) ← *linked-to*(X_1, X_3), *linked-to*(X_3, X_2).

The reason that FOIL judges the former to be more useful than the latter forms the topic of the next section.

4.3. The heuristic for evaluating literals

The description of the learning algorithm refers to a heuristic for assessing the usefulness of a literal as the next component of the right-hand side of a clause. Like ID3, FOIL uses an information-based estimate which seems to provide effective guidance for clause construction.

The whole purpose of a clause is to characterize a subset of the ⊕ tuples in a relation. It therefore seems appropriate to focus on the information provided by signalling that a tuple is one of the ⊕ kind. If the current T_i contains T_i^+ ⊕ tuples and T_i^- ⊖ tuples as before, the information required for this signal from T_i is given by

$$I(T_i) = -log_2(T_i^+ / (T_i^+ + T_i^-)).$$

If the selection of a particular literal L_i would give rise to a new set T_{i+1}, the information given by the same signal is similarly

$$I(T_{i+1}) = -log_2(T_{i+1}^+ / (T_{i+1}^+ + T_{i+1}^-)).$$

As above, suppose that T_i^{++} of the ⊕ tuples in T_i are represented by one or more tuples in T_{i+1}. The total information we have gained regarding the ⊕ tuples in T_i is given by the number of them that satisfy L_i multiplied by the information gained regarding each of them, i.e.,

$$Gain(L_i) = T_i^{++} \times (I(T_i) - I(T_{i+1})).$$

Note that this gain is negative if $\oplus$ tuples are less concentrated in T_{i+1} than in T_i, and is small if either the concentrations are similar or few $\oplus$ tuples in T_i satisfy L_i.

In the second clause of the previous network example, T_2 contains 18 $\oplus$ tuples and 54 $\ominus$ tuples, so $I(T_2)$ is 2.0. For the literal *can-reach*(X_3, X_2), ten tuples are represented in T_3, $I(T_3)$ is zero, so the gain for this literal is 20.0. The alternative literal *linked-to*(X_3, X_2), on the other hand, gives six tuples in T_3, $I(T_3)$ again being zero, for a gain of 12.0. The heuristic therefore selects the former literal over the latter.

$Gain(L_i)$ is the primary measure of the utility of a literal L_i, but it has been found desirable to assign a very small credit to an unnegated literal that introduces new free variables, even if it does not give a higher concentration of $\oplus$ tuples. The reasoning behind this is that, if no literal produces any apparent gain, it may be helpful to introduce a new variable and try again with the enlarged collection of possible literals.

4.4. Searching the space of literals

This seems a natural point to discuss the search for literals. Recall that the inner loop of FOIL builds a single clause

$$P(X_1, X_2, \ldots, X_k) \leftarrow L_1, L_2, \ldots, L_n$$

choosing at each iteration the most promising literal L_i to attach to the right-hand side. We turn now to the space of possibilities explored in the process of finding this best literal.

Each literal L_i in the right-hand side of a clause takes one of the four forms $X_j = X_k$, $X_j \neq X_k$, $Q(V_1, V_2, \ldots, V_r)$, or $\neg Q(V_1, V_2, \ldots, V_r)$, where the X_i's are existing variables, the V_i's are existing or new variables, and Q is some relation. FOIL investigates the entire space of such literals, with three significant qualifications:

- The literal must contain at least one existing variable.
- If Q is the same as the relation P on the left-hand side of the clause, possible arguments are restricted to prevent some problematic recursion.
- The form of the *Gain* heuristic allows a kind of pruning akin to alpha-beta (Winston, 1984).

The first and third of these are designed to prevent unnecessary search while the second is intended to prevent the construction of useless definitions that cause infinite recursion. The remainder of this section looks at each of these qualifications in turn.

Existing variable: FOIL requires at least one of the V_i's to be a variable introduced by the left-hand side or by a previous literal in this clause. The rationale for this restriction is that a new literal should have some linkage to previous literals in the clause.

There is a more abstract justification that depends on the order of literals in the right-hand side of a Prolog clause. Since the clause is satisfied only when all literals are satisfied,

the meaning of a clause apparently does not depend on this order, which might however affect computational efficiency. Thus the clauses

$$Q(X) \leftarrow R(X, Y), S(Y)$$
$$Q(X) \leftarrow S(Y), R(X, Y)$$

have the same meaning, but one may be faster to compute. Negated literals with variables that do not appear in the left-hand side of the clause are an exception to this general order-independence. The clause

$$Q(X) \leftarrow R(X, Y), \neg S(Y)$$

is satisfied whenever R contains a tuple $\langle X,a \rangle$, say, and $\langle a \rangle$ does not appear in the relation S. On the other hand, the clause

$$Q(X) \leftarrow \neg S(Y), R(X, Y)$$

is never satisfied if S contains any tuples at all. In this example, the right-hand side of the first clause is interpreted as

$$\exists Y\ (R(X, Y) \wedge \neg S(Y))$$

and that of the second as the very different

$$\neg (\exists Y\ S(Y)) \wedge \exists Y\ R(X, Y).$$

For a more complete discussion of the meaning of negated literals in Prolog, see Bratko (1986).

The final clause will not contain any negated literal, all of whose variable are unique to that literal, since such a literal would always be falsified. On the other hand, the order of unnegated literals does not affect the meaning of the clause, as above, so we can afford to wait until at least one of the variables in the literal has been bound.

Recursive definitions: If the relation Q of the selected literal is the same as the target relation P, some combinations of arguments of Q could lead to infinite recursion. An early version of the system attempted to ameliorate this problem by preventing a single clause calling itself with the same arguments; i.e., by ensuring that there were no values of the variables that could make $\langle X_1, X_2, \ldots, X_k \rangle$ identical to $\langle V_1, V_2, \ldots, X_k \rangle$. Unfortunately, this will not prevent unhelpful clauses such as

$$R(X_1) \leftarrow not(X_1, X_2), R(X_2)$$

in which $R(true)$ could call $R(false)$ could call $R(true)$ and so on.

The current version of FOIL is more subtle, using the notion of an *irreflexive partial ordering*, a transitive binary relation $<$ on the constants such that $x < x$ never holds for any

constant x. Consider a relation $P(X_1, X_2, \ldots, X_k)$ containing the constant tuples $\{\langle c_1, c_2, \ldots c_k\rangle\}$. A partial ordering between two arguments X_i and X_j of P is deemed to exist if the transitive closure of the relationships $\{c_i < c_j\}$ does not violate the restriction that no constant can be less than itself. Analogously, the negation of a relation can also establish a partial ordering among pairs of its arguments.

A necessary condition for a recursive relation P to be properly specified is that no invocation of P can ever lead, directly or indirectly, to its invocation with the same arguments. In this clause, $P(X_1, X_2, \ldots, X_k)$ invokes $P(V_1, V_2, \ldots V_k)$ so, if P is to avoid this problem, the pairs of tuples

$$\langle\langle X_1, X_2, \ldots, X_k\rangle, \langle V_1, V_2, \ldots, V_k\rangle\rangle$$

must satisfy some partial ordering. As a sufficient approximation, we could require one of the pairs

$$\langle X_1, V_1\rangle, \langle X_2, V_2\rangle, \ldots, \langle X_k, V_k\rangle$$

to satisfy such an ordering, which can only come from relationships among the variables established by one of the previous literals in this clause. FOIL requires a partial ordering between at least one pair $\langle X_i, V_i\rangle$ before a recursive literal will be considered for inclusion in the clause.

The second clause of the network example in Section 4.2 was

$$\textit{can-reach}(X_1, X_2) \leftarrow \textit{linked-to}(X_1, X_3), \textit{can-reach}(X_3, X_2).$$

The relation *linked-to*(A, B) is consistent with a partial ordering $A < B$; the transitive closure of $a < b$ for each pair $\langle a, b\rangle$ in the relation does not violate the above restriction. The occurrence of the literal *linked-to*(X_1, X_3) thus establishes a partial ordering $X_1 < X_3$. With this partial order established by the first literal, the recursive second literal satisfies the requirement with the first pair $\langle X_1, V_1\rangle$ above, where V_1 is now X_3. On the other hand, the literal *can-reach*(X_2, X_1) does not satisfy any such requirement—we have not established $X_1 < X_2$ or $X_2 < X_1$—and so this literal would not be considered.

This check may appear computationally expensive but such is not the case. Partial orderings between pairs of arguments of all given relations are determined only once, at the beginning of a run.[2] Thereafter it is just a matter of keeping track, for each clause, of the partial orderings that have been established so far among the variables. The check is sufficient to ensure that a clause will never cause an improper invocation of itself; the same idea can be extended to groups of clauses for the same relation.[3] However, it is still an approximation and does not cover mutually recursive definitions of different relations such as $R(c)$ invoking $S(c)$ invoking $R(c)$ for some relations R and S and constant c.

Pruning: A valuable side-effect of the form of the *Gain* heuristic discussed above is its support for significant pruning of the search space. Suppose that we have estimated the utility of an unnegated literal L_i as

$$Gain(L_i) = T_i^{++} \times (I(T_i) - I(T_{i+1})).$$

The literal L_i may contain new (free) variables and their replacement with existing (bound) variables can never increase T_i^{++}, the number of $\oplus$ tuples in T_i satisfied by L_i. Moreover, such replacement can at best produce a set T_{i+1} containing only $\oplus$ tuples, i.e., with $I(T_{i+1}) = 0$. Thus the *maximum* gain that can be achieved by any unnegated literal obtained by substituting bound variables for the free variables in L_i is

$$maximum\ gain = T_i^{++} \times I(T_i).$$

If this maximum gain is less than the best gain achieved by a literal so far, there is no need to investigate any literals obtained by such replacement. (An analogous argument can be formulated for negated literals.) Thus, for a relation Q, FOIL always gives priority to investigating literals $Q(V_1, V_2, \ldots, V_r)$ that contain many new variables and often achieves a dramatic reduction of the number of literals that must be considered.

4.5. More advanced features

This section concludes with two aspects of FOIL that go beyond the algorithm outlined in Section 4.1. The first concerns the ability to generate approximate rather than exact definitions of relations and the second involves improvement of completed clauses.

Inexact definitions: In many real-world problems it is not possible to formulate rules that precisely characterize the domain. In such circumstances, many studies have found that simple, approximate concepts can be more valuable than overspecified rules that "fit the noise." In the case of classification knowledge expressed as decision trees or as rules, improved performance on unseen cases can be obtained by "pruning" (Breiman et al., 1984; Quinlan, 1987; Clark and Niblett, 1987). In this learning context, FOIL implements *stopping criteria* for deciding

- that a clause should not be extended further, even though the current T_i is not yet free of $\ominus$ tuples, and
- that a set of clauses should not be expanded, even though they do not cover all tuples in the target relation.

These decisions are based on the perception that, for a sensible clause, the number of bits required to encode the clause should never exceed the number of bits needed to indicate explicitly the $\oplus$ tuples covered by the clause. This argument invokes Rissanen's Minimum Description Length principle (1983) and resembles that used by Quinlan and Rivest (1989).

Suppose we have a training set of size $|T|$ and a clause accounts for p $\oplus$ tuples that are in the target relation. The number of bits required to indicate those tuples explicitly is given by

$$\log_2(|T|) + \log_2\left(\binom{|T|}{p}\right)$$

On the other hand, the bits required to code a literal are

1	(to indicate whether negated)
$+\log_2$ (number of relations)	(to indicate which relation)
$+\log_2$ (number of possible arguments)	(to indicate which variables)

A clause containing n literals needs the sum of these bits, reduced by $\log_2(n!)$ since all orderings of the literals are equivalent (with the exception mentioned previously). The addition of a literal L_i to a clause is ruled out if the bits required to encode the new clause exceed the bits needed to indicate the covered tuples. If no literals can be added to a clause but it is still reasonably accurate (taken here to mean at least 85%), the inexact clause is retained as a final clause. Similarly, if no further clause can be found within this encoding restriction, the incomplete set of clauses is taken as the definition of the target relation. In either case, the user is warned that the set of clauses is imperfect.

Post-processing of clauses: Any greedy algorithm is prone to making locally optimal but globally undesirable choices. In this context, a literal chosen for the right-hand side of a clause may subsequently turn out to be unnecessary or even counterproductive. Once a clause has been developed, the literals of the right-hand side are examined to see whether any could be omitted without compromising the accuracy of the clause as a whole. Specifically, the new clause must cover all the $\oplus$ tuples covered by the old clause but must not cover additional $\ominus$ tuples. The usefulness of this simplication will be illustrated in one of the examples of Section 5.2. Similar pruning of rules in propositional domains has been found to lead to smaller, more accurate rulesets (Quinlan, 1987).

5. Results

This section presents results obtained by FOIL on a variety of learning tasks reported in the literature, thereby intending to establish that it is a powerful and general learning mechanism.

5.1. Learning family relationships

In a most interesting paper, Hinton (1986) describes a connectionist system for learning kinship relationships in two stylized isomorphic families, each with twelve members, as shown in Figure 2. He defines a network with 36 input units, three layers of hidden units, and 24 output units. The input units consist of one unit for each of the 24 individuals mentioned in the family trees and one unit for each of the twelve given relationship types: *wife*, *husband*, *mother*, *father*, *daughter*, *son*, *sister*, *brother*, *aunt*, *uncle*, *niece*, and *nephew*. Each output unit is identified with one of the 24 individuals. An input to the network consists of setting on the units associated with one individual (*B*, say) and one relationship

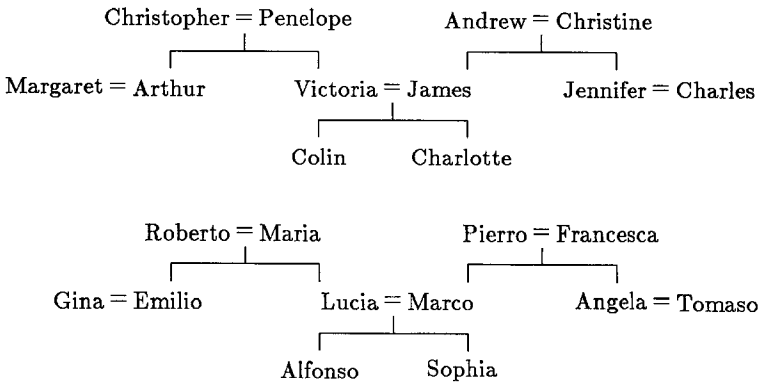

Figure 2. Two family trees, where "=" means "married to." (From Hinton, 1986.)

(R). The desired output of the network is the set of individuals $\{A\}$, each of whom has relationship R to B. For example, if the input specified the relationship *aunt* and the person *Colin*, only the output units associated with *Margaret* and *Jennifer* should be on, indicating that *Margaret* and *Jennifer* are the *aunts* of *Colin*.

The family trees in Figure 2 define 104 such input-output vector pairs. Hinton used a training set consisting of 100 of them with the remaining four pairs reserved as a test set. The network was initialized with random connection weights, allowed to learn appropriate values, and then evaluated on the four pairs in the test set. The experiment was then repeated with different initial weights. The network gave the correct output for all four of the test inputs in one trial and for three out of four in the other. (Note that, for an output to be correct, all 24 output units must have the correct on/off status; this extremely unlikely to occur by chance.)

FOIL was also given the information provided by 100 of the input-output vectors selected randomly from the 104 available but encoded as tuples rather than vectors. Each input-output pair specifies a single person B and relation R, together with the desired output status for each of the 24 people. This gives 24 tuples of the form $\langle A, B\rangle$ for the relation R, the tuple being $\oplus$ if the output unit for A should be on and $\ominus$ otherwise. Clauses were found for each relation in turn. The four input-output vectors in the test set were then used to evaluate the clauses; as before, a test vector was counted as correct only when the clauses for the relevant relation correctly predicted each of the 24 output bits. The experiment was repeated with different random divisions of the 104 relationships for a total of 20 trials.

Over these 20 trials, FOIL was correct on 78 of the 80 test cases, bettering the 7 out of 8 correct results reported by Hinton. Average time for each trial was 11 seconds on a DECstation 3100.

This learning task contains much further material for exploration, providing an excellent testbed for comparing many aspects of connectionist and symbolic learning approaches. Further experiments contrasting the approaches are being carried out and will be reported in a separate paper.

5.2. *Learning recursive relations on lists*

Numerous authors have studied the task of learning list manipulation functions, among them (Shapiro, 1981, 1983; Sammut and Banerji, 1986; Muggleton and Buntine, 1988). Four examples from the list domain should illustrate the system's performance on this kind of task.

The first example is to learn the definition of a (non-dotted) list. This task involves three relations:

list(X)	X is a list
null(X)	X is nil
components(X, H, T)	X has head H and tail T

These relations are defined for four structures

$$(),\ (a),\ (b(a)d),\ (e.f)$$

and their substructures, specifically

list:	$\{\langle() \rangle, \langle(a)\rangle, \langle(b(a)d)\rangle, \langle((a)d)\rangle, \langle(d)\rangle\}$
null:	$\{\langle() \rangle\}$
components:	$\{\langle(a), a, ()\rangle, \langle(b(a)d), b, ((a)d)\rangle,$ $\langle((a)d), (a), (d)\rangle, \langle(d), d, ()\rangle, \langle(e.f), e, f\rangle\}$

with $\ominus$ tuples provided by the closed-world asumption. From these, FOIL took less than 0.1 seconds to find the standard textbook definition (in function-free form)

$$list(A) \leftarrow components(A, B, C), list(C)$$
$$list(A) \leftarrow null(A)$$

The second task is that of finding the definition of the *member* relation. Again there are three relations:

member(X, Y)	X is a member of list Y

with *null* and *components* as before. In this example, FOIL was provided with all membership relations over the list

$$(a\ b\ (c))$$

and its subcomponents, again invoking the closed-world assumption, and the system took 0.1 seconds to find the usual definition

$$member(A, B) \leftarrow components(B, A, C)$$
$$member(A, B) \leftarrow components(B, C, D), member(A, D)$$

The astute reader will have noticed that the examples presented to FOIL for the *list* and *member* relations do not appear to have been generated randomly. In fact, they represent a simple set of examples that are somehow sufficient to discover the relations. The idea of selecting good examples from which to learn seems common in this domain and has been followed in all three systems cited above.

The next two tasks investigate the much more demanding relations

append(X, Y, Z)	appending X to Y gives Z
reverse(X, Y)	X is the reverse of Y

At the same time, any effect of selecting examples was neutralized by defining the relations over all lists of length up to four, each containing non-repeated atoms drawn from the set $\{1, 2, 3, 4\}$. This gives a total of $4! + 4 \times 3! + 6 \times 2! + 4 + 1 = 65$ lists and 4 atoms.[4] For *reverse*, the system used the closed-world assumption in which the $\ominus$ tuples consisted of all tuples not in the relation. This was not possible for *append*—there are 69^3 (or about 330,000) 3-tuples over 69 constants—so the $\ominus$ tuples were defined by selecting 10,000 of them at random. For *append* the other relations available were *null*, *list* and *components* and all of these (including *append*) were available for *reverse*. This is the same set of relations used by Shapiro (1981), even though *list* is not required for either definition.

The definition constructed for *append* in 188 seconds contains four clauses:

append(A, B, C) ← $A = C$, *null*(B)
append(A, B, C) ← $B = C$, *null*(A)
append(A, B, C) ← *components*(C, D, B), *components*(A, D, E), *null*(E)
append(A, B, C) ← *components*(C, D, E), *components*(A, D, F), *append*(F, B, E)

The second and fourth of these make up the usual definition of *append* in function-free form. The first covers a special case of a null second argument; Tom Dietterich (Private Communication, June 1989) pointed out that this additional clause is often used to improve efficiency by preventing a needless recursive unwinding of the first argument. The third clause addresses the special case of a singleton first argument, and could well be omitted.

This example also demonstrates the utility of post-processing clauses. The initial formulation of the final clause started with a literal *components*(E, G, B) which effectively limited the clause to cases where the first argument is a two-element list. When the clause had been completed, the post-processing noted that this literal could be omitted, resulting in the perfectly general clause shown.

From the data for *reverse*, FOIL required 247 seconds to find the definition

reverse(A, B) ← $A = B$, *null*(A)
reverse(A, B) ← $A = B$, *append*(A, C, D), *components*(D, E, C)
reverse(A, B) ← *append*(C, D, A), *append*(D, E, B), *components*(B, F, E), *reverse*(C, E)

Again, the second clause is strictly unnecessary, being a rather convoluted way of expressing the fact that a single-element list is its own reverse.

The times required to find the definitions of these relations are much greater than those reported by Shapiro (1981), namely 11 and 6 seconds respectively on a DEC 2060. For these tasks, however, there has been no attempt to construct a minimal set of tuples from which suitable definitions can be found. FOIL is working with 10,261 tuples for *append* and 4,761 for *reverse* compared to 34 and 13 respectively in the case of Shapiro's MIS system. On the other hand, FOIL is probably incapable of finding adequate definitions from such small numbers of tuples. Whereas MIS's search for clauses is guided and constrained by external information (e.g., it can ask questions), FOIL navigates by extracting information-based hints from the data. The *Gain* heuristic is essentially a statistical measure; as clauses become more complex, sensible choice of the first literals depends on being able to distinguish helpful clues from simple random variations and this is harder if the number of tuples in the training set is small.

5.3. Learning the concept of an arch

Winston (1975) describes the task of learning the nature of an arch from four objects, two of which are arches and two not, as shown in Figure 3. The domain involves several relations:

arch(A, B, C)	A, B and C form an arch with lintel A
supports(A, B)	A supports B
left-of(A, B)	A is left of B
touches(A, B)	the sides of A and B touch
brick(A)	A is a brick
wedge(A)	A is a wedge
parallelpiped(A, B)	A is a brick or a wedge

which are used to describe the four objects and their 12 components. FOIL takes 1.2 seconds to find the definition

$$arch(A, B, C) \leftarrow left\text{-}of(B, C), supports(B, A), \neg touches(B, C)$$

It is interesting to note the difference between this definition and the concept enunciated by Winston's program. Since FOIL looks for concepts in a general-to-specific fashion, it only discovers descriptions that are minimally sufficient to distinguish tuples in the relation from other tuples, which Dietterich and Michalski (1981) refer to as *maximally general*

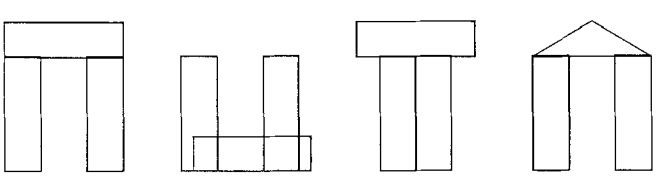

Figure 3. Arches and near misses. (From Winston, 1975.)

descriptions. In this case, FOIL never formulates a requirement that the lintel be either a block or a wedge, or that both sides support it, because it has never seen 'near misses,' in Winston's terms, that make these properties relevant. FOIL would require a description of many more objects in order to elaborate its definition of the relation *arch*.

5.4. Learning where trains are heading

In a paper introducing his INDUCE system, Michalski (1980) describes an artificial task of learning to predict whether a train is headed east or west. This task illustrates the kind of structured objects with varying numbers of substructures that cause problems for attribute-value representation; the ten trains of Figure 4 have different numbers of cars and some cars carry more than one load. Five of the trains are heading east and five west. This time there is a plethora of relations:

eastbound(T)	train T is eastbound
has-car(T, C)	C is a car of T
infront(C, D)	car C is in front of D
long(C)	car C is long
open-rectangle(C)	car C is shaped as an open rectangle
...	similar relations for five other shapes
jagged-top(C)	C has a jagged top
sloping-top(C)	C has a sloping top
open-top(C)	C is open
contains-load(C, L)	C contains load L
1-item(C)	C has one load item
...	similar relations for two and three load items
2-wheels(C)	C has two wheels
3-wheels(C)	C has three wheels

From the 180-odd tuples in these relations, FOIL takes 0.2 seconds to find the definition

$$eastbound(A) \leftarrow has\text{-}car(A, B),\ \neg\, long(B),\ \neg\, open\text{-}top(B)$$

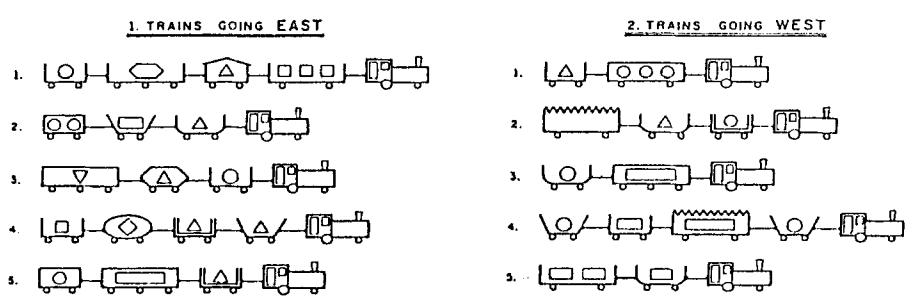

Figure 4. Ten trains. (From Michalski, 1980.)

which is the same concept found by INDUCE. If the problem is turned around by replacing *eastbound* by *westbound*, FOIL finds the clause

$$westbound(A) \leftarrow has\text{-}car(A, B), long(B), 2\text{-}wheels(B), \neg open\text{-}top(B)$$

that explains three of the five westbound trains, but the encoding length heuristics prevent it discovering another clause. INDUCE, on the other hand, uses its ability to generate new descriptors, here counting the cars (including the engine) in each train. It then finds a rule which might be expressed by the clauses

$$westbound(A) \leftarrow car\text{-}count(A) = 3$$
$$westbound(A) \leftarrow has\text{-}car(A, B), jagged\text{-}top(B).$$

5.5. *Chess endgame*

Muggleton, Bain, Hayes-Michie, and Michie (1989) describe a learning task that nicely demonstrates the utility of being able to generate imperfect but accurate definitions. The domain is the chess endgame White King and Rook versus Black King. The target relation is the six-place predicate *illegal*(A, B, C, D, E, F) indicating that the position in which the White King is at (A, B), the Rook at (C, D), and the Black King at (E, F) is not a legal Black-to-move position. There are two other relations available:

$adj(X, Y)$	row or column X is adjacent to Y
$less\text{-}than(X, Y)$	row or column X is less than Y

A training set for this task consists of a random sample of positions, some of which (about 34%) are illegal and some not. Experiments were carried out with training sets containing 100 positions and 1000 positions. The learned definition was then tested on a large random sample of positions selected independently to the training sets. Note that there is no simple correct definition of *illegal* in terms of the given relations. The amount of information provided by training sets of this size is probably insufficient to allow any learning system to discover an exact set of clauses.

Table 1 shows results reported by Muggleton et al. for two systems, CIGOL and DUCE, that learn clauses. Each experiment was performed five times; the accuracies shown are the averages obtained over 5,000 unseen cases and the approximate times are for a Sun

Table 1. Results on the chess endgame task.

	100 training objects		1000 training objects	
System	Accuracy	Time	Accuracy	Time
CIGOL	77.2%	21.5 hr	N/A	N/A
DUCE	33.7%	2 hr	37.7%	10 hr
FOIL	92.5% sd 3.6%	1.5 sec	99.4% sd 0.1%	20.8 sec

3/60. The table also gives similar results for FOIL averaged over 10 repetitions; times are for a DECstation 3100 and accuracies were determined over 10,000 unseen positions. FOIL finds more accurate definitions than CIGOL or DUCE, and requires much less time than can be accounted for by differences in hardware and implementation language.

As an example, the clauses found by FOIL on the first trial with 100 training objects are

$$\begin{aligned}
&\textit{illegal}(A, B, C, D, E, F) \leftarrow C = E\\
&\textit{illegal}(A, B, C, D, E, F) \leftarrow D = F\\
&\textit{illegal}(A, B, C, D, E, F) \leftarrow \textit{adj}(B, F), \textit{adj}(A, E)\\
&\text{illegal}(A, B, C, D, E, F) \leftarrow A = E, B = F\\
&\textit{illegal}(A, B, C, D, E, F) \leftarrow A = C, B = D
\end{aligned}$$

These may be paraphrased as: a position is illegal if the Black King is on the same row or column as the Rook, or the White King's row and column are both next to the Black King's, or the White King is on the same square as the Black King or the Rook. The clauses are certainly not exact—they cover only 30 of the 32 *illegal* tuples in this training set—but they correctly classify better than 95% of the 10,000 unseen cases.

5.6. *Eleusis*

The final task concerns the card game Eleusis in which players attempt to learn rules governing sequences. The dealer invents a secret rule specifying the conditions under which a card can be added to a sequence of cards. The players attempt to add to the current sequence; each card played is either placed to the right of the last card in the sequence if it is a legal successor, or placed under the last card if it is not. The horizontal *main line* thus represents the sequence as developed so far while the vertical *side lines* show incorrect plays.

Figure 5 contains three game layouts taken from Dietterich and Michalski (1986), all of which arose in human play. The same paper describes SPARC/E, a system that tries to discover the dealer's rule from given layouts, i.e., the system learns from passive observation rather than experimentation. SPARC/E handled these problems well, producing a few hypotheses that included a reasonable rule in each case, and requiring only 0.5 to 6.5 seconds on a CYBER 175 to do so.

The same problems were presented to FOIL. After some experimentation, the following relations were set up to capture information noted by Dietterich and Michalski that is relevant to these examples:

can-follow(A, B, C, D, E, F)	card A (suit B) can follow a sequence ending with • card C (suit D), • E consecutive cards of suit D, and • F consecutive cards of the same color
precedes(X, Y)	suit/rank X precedes suit/rank Y
lower(X, Y)	suit/rank X is lower than suit/rank Y
face(X)	rank X is a face card
same-color(X, Y)	suits X and Y are the same color
odd(X)	rank/length X is odd

main line	A♡	7♣	6♣	9♠	10♡	7♡	10♢	J♣	A♢	4♡	8♢	7♣	9♠
side lines			K♢			5♠		Q♢		3♠			9♡
			J♡										6♡
main line (ctd)	10♣	K♠	2♣	10♠	J♠								
side lines (ctd)		Q♡											
		A♢											

main line	J♣	4♢	Q♡	3♠	Q♢	9♡	Q♣	7♡	Q♢	9♢	Q♣	3♡	K♡
side lines	K♣	5♠				4♠		10♢					
		7♠											
main line (ctd)	4♣	K♢	6♣	J♢	8♢	J♡	7♣	J♢	7♡	J♡	6♡	K♢	

main line	4♡	5♢	8♣	J♠	2♣	5♠	A♣	5♠	10♡
side lines	7♣	6♠	K♣	A♡		6♣		A♠	
	J♡	7♡	3♡	K♢					
	4♣	2♣		Q♠					
	10♠	7♠							
	8♡	6♢							
	A♢	6♡							
	2♢	4♣							

Figure 5. Three Eleusis layouts. (From Dietterich and Michalski, 1986.)

Each card other than the first in the sequence gives a tuple for the target *can-follow* relation, ⊕ if the card appears in the main line and ⊖ if it is in a side line.

For the first layout, FOIL found the clauses

$$can\text{-}follow(A, B, C, D, E, F) \leftarrow same\text{-}color(B, D)$$
$$can\text{-}follow(A, B, C, D, E, F) \leftarrow odd(F), odd(A)$$

in 0.1 seconds. The rule, which can be paraphrased "completed color sequences must be odd and must end with an odd card," does not account for one card in the main line. The rule intended by the dealer, "completed color sequences must be odd, and a male card cannot appear next to a female card," uses information on card sex that is not encoded in the above relations. Interestingly, SPARC/E also finds the inexact rule "completed sequences must be odd" which errs the other way in being too general.

The second layout gave

$$can\text{-}follow(A, B, C, D, E, F) \leftarrow face(A), \neg face(C)$$
$$can\text{-}follow(A, B, C, D, E, F) \leftarrow face(C), \neg face(A)$$

("play alternate face and non-face cards," the rule intended by the dealer) in 0.2 seconds.

The third layout presented a different problem. FOIL found the single clause

$$can\text{-}follow(A, B, C, D, E, F) \leftarrow precedes(B, D), \neg lower(A, C)$$

("play a higher card in the suit preceding that of the last card") in 0.2 seconds, but the encoding heuristics judged that the layout contained insufficient information to support another clause. The system does not discover the other half of the rule ("play a lower card in the suit following that of the last card") unless the main line is extended with two more cards.

Both SPACE/E and FOIL are quite competent in this domain, but they differ in one important respect. Whereas SPACE/E generates several hypotheses, some of which may be too complex to be justified by the given layout, FOIL plumps for a single hypothesis which may be incomplete if the layout is small.

5.7. On evaluation

This section has discussed FOIL's performance on examples from six learning domains. In only two of these, the family and chess endgame tasks, have the learned rules been evaluated by the acid test of using them to classify unseen cases. While this is the most convincing proof that useful learning has occurred, it depends on the existence of a substantial body of examples from which to extract training and test sets, and so is ruled out for tasks like *arch* and *trains*. The purpose in these other sections has been to show that FOIL can discover exact definitions (as in the list manipulation domain), or can find concepts similar to those produced by other learning systems when given similar input.

6. Limitations of FOIL

The previous section notwithstanding, it is not difficult to construct tasks on which the current version[5] of FOIL will fail. As a simple illustration, consider the relations

P: $\{\langle 1\rangle, \langle 2\rangle\}$
A: $\{\langle 1, t\rangle, \langle 2, f\rangle, \langle 3, t\rangle, \langle 4, f\rangle\}$
B: $\{\langle 1, t\rangle, \langle 2, t\rangle, \langle 3, f\rangle, \langle 4, f\rangle\}$
C: $\{\langle 1, f\rangle, \langle 2, t\rangle, \langle 3, f\rangle, \langle 4, t\rangle\}$
Q: $\{\langle t\rangle\}$

When we are looking for the first literal of the first clause for the target relation P, the three literals $A(X_1, X_2)$, $B(X_1, X_2)$, and $C(X_1, X_2)$ have identical utilities as measured by the *Gain* heuristic, but only one of them (the second) leads to any definition for P. There is no way that FOIL can pick the winner. Having chosen the first, however, FOIL paints itself into a corner—there is no completion of the clause that gives an acceptably accurate rule—so the search fails. The same lack of lookahead can make FOIL quite sensitive to the way relations are formulated for a task.

This highlights the most glaring weakness of the system, namely the greedy search used to assemble clauses from literals. FOIL carries out a quasi-exhaustive search of combinations of variables when evaluating a possible literal $Q(V_1, V_2, \ldots, V_r)$ but, having selected

one, the search for a clause does not explore any alternatives. The former search is usually not expensive as the number of possible values for each V_i is small (since V_i is either one of the variables already bound in this clause or a new variable), and the pruning discussed earlier is quite effective. A similar exhaustive search is far too expensive for the stepwise construction of clauses, but two alternatives to greedy search suggest themselves:

- *Beam search:* In the style of AQ and CN2, this strategy retains the best N partial structures at each step, for some fixed N, and so reduces the global impact of a single poorly-chosen literal. The advantage of beam search is that it increases search effort by just a constant factor N.
- *Checkpoints:* In this approach, choice points at which two or more alternatives appear to be roughly equal are flagged. If the greedy search fails, the system reverts to the most recent checkpoint and continues with the next alternative.

Future versions of FOIL will incorporate one of these more sophisticated search paradigms in an attempt to overcome the short-sightedness discussed above without incurring an exponential increase in computational cost.

Another present limitation arises from the lack of continuous values, since all constants are assumed to be discrete. It appears likely that this deficiency can easily be overcome by allowing typed constants and literals of the form $X_i > t$, where X_i is a variable with real values and t is some threshold. Techniques such as that described by Breiman et al. (1984) enable attribute-value systems to make perceptive choices of suitable thresholds, and there seems no reason why the same techniques should not function equally well in FOIL's environment.

The restriction of definitions to Horn clause logic is another weakness, this time a fundamental one. Although the language is sufficient for many learning tasks, it is a proper subset of first-order logic and cannot express some concepts. In the blocks world studied by Vere (1978), for instance, a rule describing a legal robot action needs to express the idea that one block is moved but all other blocks remain unchanged. In first-order logic we might write something like

$$X \textit{ is moved} \wedge \forall Y\ (Y \neq X \rightarrow Y \textit{ is not moved})$$

but there is no way to express this in Horn clause logic using the same predicates because all variables on the right-hand side of a clause are existentially, not universally, quantified.

As noted earlier, the methods used by FOIL to avoid infinitely recursive definitions are incomplete. The system can currently guarantee that no single clause will cause a problem in this respect, and the same ideas can be extended to all clauses for a particular relation. This seems to permit the discovery of many common recursive definitions, but it is still only an approximation. It should be possible to develop more flexible criteria, perhaps by empirical testing of definitions to see whether trial cases cause problems in this respect.

This section has highlighted some of the deficiencies of FOIL. In the next and final section, the strengths and weaknesses of the system are examined in comparison with other learning approaches that tackle similar tasks.

7. Discussion

FOIL was originally conceived as a vehicle for adapting the simple attribute-value learning techniques of Section 2 to Hinton's kinship learning task described in Section 5.1. Along the way it was extended to the general learning method discussed in this paper. It seems appropriate to conclude by comparing it to some other symbolic learning systems that produce first-order rules.

Two key ideas underpinning many such systems are *proof* and *maximally specific generalization*. Shapiro's (1981, 1983) MIS, for example, constructs theories incrementally from specific facts. When it encounters a fact that is inconsistent with the current theory, MIS walks back up the proof tree that establishes the contradiction and frames questions about other specific facts that allow it to isolate the flaws in the theory. Proof also is the central operation in all explanation-based learners (Mitchell et al., 1986; DeJong and Mooney, 1986). On the other hand, systems like SPROUTER (Hayes-Roth and McDermott, 1977) and THOTH-P (Vere, 1978) find all properties common to a collection of examples of a concept and use the conjunction of these properties as a general description of the concept. Such an approach makes it possible to learn from only positive examples of a concept; Dietterich and Michalski (1981) provide an analytical review of this use of generalization. Anderson and Kline's (1979) ACT also uses maximally specific generalization of two examples represented as a conjunction of properties, although ACT also uses feedback on incorrect overgeneralizations to debug faulty learned concepts; in this sense it resembles MIS, but the correction is heuristic rather than proof-based.

FOIL has no notion of proof—the validity of a clause is investigated by looking for specific counter-examples in the training set. Moreover, the system looks for maximally general descriptions that allow positive examples to be differentiated from negative ones, rather than maximally specific descriptions that express all shared properties of subsets of positive examples. FOIL cannot learn anything from $\oplus$ tuples alone and needs either explicit $\ominus$ tuples or the closed-world assumption.

Langley's (1985) SAGE.2 uses discrimination as the mechanism for learning first-order production rules. The domain is state-space problem-solving in which a sequence of transformations is sought which will change an initial state to one satisfying a goal. The role of SAGE.2's rules is to suggest transformations in a given *context*, i.e., the current state and information about how it was obtained from the initial state. When a transformation suggested by a rule R is subsequently found to be inappropriate, SAGE.2 examines this *rejection* context and the most recent *selection* context in which the rule's recommendation was useful. A new rule is constructed that adds to the antecedent of R some logical expression satisfied by the selection context but not by the rejection context. New variables are introduced to replace constants (in clause terminology), so that the new rule will be useful in other contexts.

SAGE.2 thus constructs a rule from an existing rule and two (ground) instances of its use, and does so in one step. ACT (Anderson and Kline, 1979) likewise constructs a first-order rule from two ground instances. If the instances are of the same class, ACT generalizes by introducing variables where the instances have different constants, imposing a limit on the proportion of constants that can be replaced. If the instances belong to different classes, the system discriminates by substituting a constant for one existing variable, or adding to the rule antecedent conditions that limit the possible values of one variable. Both these

systems thus use two specific instances to suggest the form of a new or revised rule and what variables the rule should contain. This is a very different approach to that adopted by FOIL in which a much larger space of possible rules is explored by adding literals one at a time, looking at combinations of existing and new variables at each step, but guided by all relevant instances rather than just two.

Two other systems, KATE (Manago, 1989) and GARGANTUBRAIN (Brebner, 1988), also extend empirical learning methods to the discovery of first-order rules. KATE represents information about objects in frames and produces an extended decision tree in which a test at an internal node may involve a new, existentially quantified variable. GARGANTUBRAIN produces general first-order descriptions of positive examples, themselves represented by ground Horn clauses. The rule language allows the negation of arbitrary logical expressions to appear in the right-hand side of a clause, a useful extension of standard Horn clause logic. However, neither system can find recursive definitions, a limitation that also applies to most systems whose primary operation is generalization, with the exception of MARVIN (Sammut and Banerji, 1986).

Systems such as SPROUTER (Hayes-Roth and McDermott, 1977), MIS (Shapiro, 1981, 1983), Winston's (1975) structural learning program, and MARVIN (Sammut and Banerji, 1986) are *incremental* learners; they can modify what has been learned in the light of new objects. FOIL requires all tuples for a relation to be available before any clause is generated. However, Schlimmer and Fisher (1986) showed that the non-incremental ID3 could be transformed into the quasi-incremental $\widehat{\text{ID3}}$ by the following strategy: if the current rule performs satisfactorily on the current training instance, leave the rule unchanged; otherwise, form a new rule by invoking ID3 on all the training instances received to date. A similar approach might be possible with FOIL.

Some systems, such as INDUCE (Michalski, 1980) and CIGOL (Muggleton and Buntine, 1988), can suggest new predicates that simplify the definitions they are constructing. FOIL does not have any mechanism of this kind, since it is restricted to finding a definition of a target relation in terms of the available relations. However, the techniques for inventing new predicates that have been developed in such systems might perhaps be grafted onto FOIL, thereby expanding its vocabulary of literals and assisting its search for simple clauses.

Learning logical defnitions requires the exploration of a very large space of theory descriptions. Authors such as Shapiro (1981, 1983), Sammut and Banerji (1986), and Muggleton and Buntine (1988) finesse this problem by postulating the existence of an oracle that answers questions posed by their systems. In Shapiro's MIS this guidance includes specifying the validity of ground facts, while in MARVIN and CIGOL it is confirmation of the correctness of proposed generalizations. In either case, the existence of such an oracle can prevent the systems' setting off down fruitless paths. FOIL, on the other hand, uses only the information represented by the given relations in the form of sets of constant tuples. Conversely, FOIL will usually require more data than these systems, since it is dependent on information implicit in this data to guide search.

None of the current systems for constructing first-order rules seem to have addressed the question of inexact clauses, although this is a common concern in empirical systems using attribute-value descriptions, as discussed, for example, by Breiman et al. (1984). In this respect, FOIL seems to be unique.

FOIL finds definitions of relations one at a time, using other relations as a kind of background knowledge. In this sense, FOIL is much more restricted than explanation-based learners (Mitchell et al., 1986; DeJong and Mooney, 1986) in the way background knowledge can be expressed. Although it would be possible to allow a background relation to be represented by an arbitrary program that determines whether a given constant tuple is in the relation (Pat Langley, Personal Communication, June 1989), this would require considerably more computation than the current algorithm's use of explicit tuples.

In summary, then FOIL has several good points: it

- uses efficient methods adapted from attribute-value learning systems
- can find recursive definitions
- can develop inexact but useful rules
- does not require an oracle.

On the negative side, the system

- is restricted to rules expressible in function-free Horn clauses
- cannot postulate new predicates
- requires a training set for the target relation that contains both $\oplus$ and $\ominus$ tuples
- is not incremental
- is based on a short-sighted, greedy algorithm.

Future extensions will address at least the last of these issues. Even in its present state, FOIL can cope with a wide range of learning tasks that have been investigated with other systems. The goal of this research is to develop a system capable of handling learning tasks of practical significance. For example, FOIL's input consists of constant tuples in relations, the same form as that used for storing information in relational databases. One application for an improved FOIL would be to discover regularities, expressed as clauses, in a large volume of real-world information.

Acknowledgments

I gratefully acknowledge support for this research from the Australian Research Council. Thanks to many colleagues who have helped me to refine the ideas presented here, especially Pat Langley, Norman Foo, Dennis Kibler, Jack Mostow, and Tom Dietterich.

Notes

1. Since the clause-generating process does not make use of clauses already found for other relations, the order in which relations are considered is immaterial.
2. The cost of ruling out a partial ordering between two arguments of a relation is at most proportional to the square of the number of constants; in practice, it is very small.
3. At a cost of ensuring that the different partial orderings used by different clauses in the group are mutually consistent.

4. The current experimental coding of FOIL is limited to 128 constants and so is unable to accommodate all lists of length up to five, or any greater number.
5. The present FOIL, which should perhaps be called FOIL.0, is a 2300-line program written in C. It incorporates some niceties (such as the construction of relation indexes to permit an efficient *join* operation) but suffers from many annoying restrictions (e.g., constants must be denoted by a single character). The code, such as it is, is available on request.

References

Anderson, J.R., and Kline, P.J. (1979). A learning system and its psychological implications. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, (pp. 16–21). Tokyo, Japan.

Bratko, I. (1986). *Prolog programming for artificial intelligence*. Wokingham: Addison-Wesley.

Brebner, P.C. (1988). *Gargantubrain: An heuristic algorithm for learning non-recursive Horn clauses from positive and negative examples*. (Technical Report 8802). Sydney, Australia: University of New South Wales, School of Electrical Engineering and Computer Science.

Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J. (1984). *Classification and regression trees*. Belmont: Wadsworth.

Cestnik, B., Kononenko, I., and Bratko, I. (1987). ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In I. Bratko and N. Lavrač (Eds.), *Progress in machine learning*. Wilmslow: Sigma Press.

Clark, P., and Niblett, T. (1987). Induction in noisy domains. In I. Bratko and N. Lavrač (Eds.), *Progress in machine learning*. Wilmslow: Sigma Press.

Clark, P., and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*, 261–284.

DeJong, G., and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*, 145–176.

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

Dietterich, T.G., and Michalski, R.S. (1986). Learning to predict sequences. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol 2). Los Altos: Morgan Kaufmann.

Hayes-Roth, F., and McDermott, J. (1977). Knowledge acquisition from structural descriptions. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 356–362). Cambridge, MA: Morgan Kaufmann.

Hinton, G.E. (1986). Learning distributed representations of concepts. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, MA: Lawrence Erlbaum.

Hunt, E.B., Marin, J., and Stone, P.J. (1966). *Experiments in induction*. New York: Academic Press.

Langley, P. (1985). Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science*, *9*, 217–260.

Manago, M. (1989). Knowledge intensive induction. In *Proceedings of the Sixth International Machine Learning Workshop*. Ithaca, NY: Morgan Kaufmann.

Michalski, R.S. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *2*, 349–361.

Michalski, R.S., Mozetič, I. Hong, J., and Lavrač, N. (1986). The multipurpose incremental learning system AQ15 and its testing application to three medical domains. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 1041–1045). Philadelphia, PA: Morgan Kaufmann.

Mitchell, T.M., Keller, R.M., and Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*, 47–80.

Muggleton, S., and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 339–352). Ann Arbor, MI: Morgan Kaufmann.

Muggleton, S., Bain, M., Hayes-Michie, J., and Michie, D. (1989). An experimental comparison of human and machine learning formalisms. *Proceedings of the Sixth International Machine Learning Workshop* (pp. 113–188). Ithaca, NY: Morgan Kaufmann.

Quinlan, J.R. (1979). Discovering rules by induction from large collections of examples. In D. Michie (Ed.), *Expert systems in the micro electronic age*. Edinburgh: Edinburgh University Press.

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

Quinlan, J.R. (1987). Simplifying decision trees. *International Journal of Man-Machine Studies, 27*, 221-234.

Quinlan, J.R. (1988). Decision trees and multi-valued attributes. In J.E. Hayes, D. Michie and J. Richards (Eds.), *Machine Intelligence 11*. Oxford: Oxford University Press.

Quinlan, J.R., and Rivest, R.L. (1989). Inferring decision trees using the Minimum Description Length principle. *Information and Computation, 80*, 227-248.

Rissanen, J. (1983). A universal prior for integers and estimation by minimum description length. *Annals of Statistics, 11*, 416-431.

Rivest, R.L. (1988). Learning decision lists. *Machine Learning, 2*, 229-246.

Sammut, C.A., and Banerji, R.B. (1986). Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), *Machine Learning: An artificial intelligence approach* (Vol 2). Los Altos: Morgan Kaufmann.

Schlimmer, J.C., and Fisher, D. (1986). A case study of incremental concept formation. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 296-501). Philadelphia, PA: Morgan Kaufmann.

Shapiro, E.Y. (1981). An algorithm that infers theories from facts. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 446-451). Vancouver, BC: Morgan Kaufmann

Shapiro, E.Y. (1983). *Algorithmic program debugging*. Cambridge, MA: MIT Press.

Vere, S.A. (1978). Inductive learning of relational productions. In D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern-directed inference systems*, New York: Academic Press.

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

Winston, P.H. (1984). *Artificial intelligence* (2nd Ed.). Reading: Addison-Wesley.