Appears in *New Generation Computing 13 (1995) 287-312*

Induction of Logic Programs: FOIL and Related Systems

J. R. Quinlan and R. M. Cameron-Jones

J. R. Quinlan
University of Sydney
Sydney Australia 2006
quinlan@cs.su.oz.au

R. M. Cameron-Jones
University of Tasmania
Launceston Australia 7250
Michael.CameronJones@appcomp.utas.edu.au

Abstract: FOIL is a first-order learning system that uses information in a collection of relations to construct theories expressed in a dialect of Prolog. This paper provides an overview of the principal ideas and methods used in the current version of the system, including two recent additions. We present examples of tasks tackled by FOIL and of systems that adapt and extend its approach.

1. Introduction

All symbolic machine learning leads to the formulation or modification of theories, so the language in which theories are expressed is an important consideration. First-order theory languages have been used for at least thirty years, as documented by Sammut [1993]. Explanation-based generalisation systems [Mitchell, Keller and Kedar-Cabelli, 1986; DeJong and Mooney, 1986] have always required them, but the early and influential work of Shapiro [1983] and Sammut and Banerji [1986] also employed them in an inductive learning context. Nevertheless, first-order empirical learning, including what we now call inductive logic programming, did not attract widespread attention until the 1990s.

Training data in zeroth-order learning consists of attribute-value vectors, each belonging to a known class. Theories are propositional functions from attribute values to classes and are expressed in forms such as decision trees [Quinlan, 1993]. In first-order learning, training data comprises a *target* relation, defined extensionally as a set of tuples of ground terms, and a set of *background* relations that might be defined extensionally or intensionally. The goal of learning is to construct a logic program that constitutes an intensional definition of the target relation in terms of itself and the background relations. Such theories permit recursion and limited quantification, both advantageous when dealing with structured objects that are difficult to describe in attribute-value form. Where zeroth-order learning refers to examples and counter-examples of some concept, first-order learning refers analogously to tuples belonging, or not belonging, to the target relation. Since this is somewhat long-winded, we refer to such tuples here as $\oplus$ or $\ominus$ tuples respectively.

We say that a (complete) theory *covers* a tuple if the corresponding ground query to the logic program succeeds. The goal of first-order learning can thus be stated as the construction of a theory that covers all $\oplus$ tuples and no $\ominus$ tuples of the target relation. During learning, however, when only a partial theory exists, this definition of "covers" might be bent slightly; for instance, a recursive literal might be evaluated by lookup in the extensional definition of the target relation rather than by attempting to execute the incomplete program.

First-order learning systems can be grouped into two families. Most earlier systems such as MIS [Shapiro, 1983], MARVIN [Sammut and Banerji, 1986] , and CIGOL [Muggleton and Buntine, 1988] are based on the successive revision method. A faulty theory is too general if it covers a $\ominus$ tuple and too specific if it fails to cover a $\oplus$ tuple. When a new tuple is treated erroneously, the query computation is examined, perhaps with the help of an oracle, to pinpoint the defect in the theory that is responsible for the error. The theory is revised accordingly and the process continues with the next tuple. This style of learning falls within the *identification in the limit* paradigm [Gold, 1967] in which it is often possible to prove that systems will converge on a correct theory after seeing sufficient training tuples. In practice, though, this family of algorithms is computationally demanding and is effectively limited to tasks that involve a small number of carefully chosen examples.

The other family uses instead a separate-and-conquer strategy pioneered by Michalski [1980]. All training tuples are considered together and, at each iteration, a clause of the theory is found that covers some $\oplus$ tuples but no $\ominus$ tuples. The covered tuples are then discarded and the process iterates until all $\oplus$ tuples are covered by at least one clause. The family is further subdivided by the method used to find a suitable clause. Top-down systems such as FOIL [Quinlan, 1990] start with a general clause head and add literals to the body until all $\ominus$ tuples are excluded. Bottom-up systems exemplified by GOLEM [Muggleton and Feng, 1992] form a most specific generalisation of a small subset of the $\oplus$ tuples, then generalise this further by dropping literals so long as the clause covers no $\ominus$ tuples. Both bottom-up and top-down systems have successfully tackled large-scale tasks and have proven to be orders of magnitude faster than systems based on successive revision.

This paper focusses on FOIL, an early member of the top-down group. We describe the learning task in more detail and outline key features of the system. Many of these have been reported previously (the best general references being [Quinlan, 1990] and [Cameron-Jones and Quinlan, 1994]) and so are only sketched here, but two more recent additions to FOIL are treated at greater length. Examples of tasks investigated with FOIL, two of which are new, are presented. Numerous other systems have adapted or extended elements of FOIL's approach and several of these related systems are reviewed. We finish with some areas for further research.

2. Description of FOIL

As mentioned above, input to FOIL includes information about relations. In common with many (but not all) first-order learning systems, FOIL requires the target and all background relations to be defined extensionally by sets of tuples of constants. Every relation argument has a specified type; there may be many distinct types or all constants can be regarded as belonging to a single type. Although the intensional definition learned from these extensional relations is derived from a particular set of examples, it is intended to be executable as a Prolog program in which the background relations may also be specified intensionally by definitions rather than by sets of ground tuples. For example, FOIL might learn a definition of *even integer* from just the integers in [0,10] and background relations defined over these integers, but the learned definition, when used with intensional background relations, should be capable of deciding whether an arbitrary integer is even.

The language in which FOIL expresses theories is a restricted form of Prolog that omits cuts, fail, disjunctive goals, and functions other than constants. This last does not pose any particular problem since Prolog programmers are accustomed to defining functions by relations; a k-ary function can be represented by a $k+1$-argument relation in which the last argument gives the value of the function applied to the first k arguments. Negated literals *not(L(...))* are permitted, where *not* is interpreted as negation by failure as in Prolog.

As an example of a task, consider learning a definition of the membership relation on lists from a small world containing just the lists [ ], [1], [2], [3], [1,2], [2,3], and [1,2,3]. The target relation member(E,L) contains pairs whose first constant denotes an element that belongs in the list denoted by the second. In this small world there are just ten tuples in member:

$$\begin{array}{ccccc} \langle 1,[1]\rangle & \langle 2,[2]\rangle & \langle 3,[3]\rangle & \langle 1,[1,2]\rangle & \langle 2,[1,2]\rangle \\ \langle 2,[2,3]\rangle & \langle 3,[2,3]\rangle & \langle 1,[1,2,3]\rangle & \langle 2,[1,2,3]\rangle & \langle 3,[1,2,3]\rangle \end{array}$$

the first denoting that element 1 is a member of the list [1] and so on. As far as FOIL is concerned, lists like [1,2,3] are just constants, so a background relation components(L,H,T) is required to show how to find the head H and tail T of a list L. The tuples making up components are

$$\langle [1],1,[\,]\rangle \quad \langle [2],2,[\,]\rangle \quad \langle [3],3,[\,]\rangle \quad \langle [1,2],1,[2]\rangle \quad \langle [2,3],2,[3]\rangle \quad \langle [1,2,3],1,[23]\rangle$$

where the first states that list [1] has head 1 and tail [ ].

All the tuples that belong to the relation member are clearly $\oplus$ tuples. The corresponding $\ominus$ tuples needed by FOIL can be provided explicitly or, more commonly, can be determined using the *closed world* assumption. That is, all tuples consisting of an element and a list as above that do not appear explicitly in the relation member can be assumed not to belong to the relation, implying that $\langle 1,[\,]\rangle$, $\langle 1,[2]\rangle$, $\langle 3,[1,2]\rangle$ and so on are all $\ominus$ tuples. The number of such $\ominus$ tuples may be overwhelming when the target relation has high arity, so FOIL contains an optional facility to use only a random sample of them.

Initialisation:
theory *:= null program*
remaining *:= all* $\oplus$ *tuples of target relation R*

While remaining *is not empty*

/ Grow a new clause */*

clause *:=* $R(A, B, ...)$ *:-*

While clause *covers* $\ominus$ *tuples of R*

Find appropriate literal(s) L (e.g. to exclude some $\ominus$ *tuples)*
Add L to right-hand side of clause

Remove $\oplus$ *tuples covered by* clause *from* remaining
Add clause *to* theory

Figure 1: Outline of FOIL

2.1 Overview of the learning algorithm

As outlined in Figure 1, FOIL uses the separate-and-conquer method, iteratively learning a clause and removing the $\oplus$ tuples that it covers until none remain. A clause is grown by successive specialisation, starting with the most general clause head and adding literals to the body until the clause does not cover any $\ominus$ tuples.

Clause construction is guided by the bindings of the variables in the partial clause that satisfy the clause body. If the clause contains k variables, a binding is a k-tuple of constants that specifies the value of all variables in order. Each such possible binding is labelled $\oplus$ or $\ominus$ according to whether the tuple of values for the variables in the clause head does or does not belong in the target relation.

We illustrate the process using the member relation. The initial clause consists of just the head

member(A,B) :-

in which each variable is unique. The labelled bindings corresponding to this initial partial clause are just the $\oplus$ and $\ominus$ tuples of the target relation, namely

$$
\begin{array}{rrrrr}
\langle 1,[1]\rangle\oplus & \langle 2,[2]\rangle\oplus & \langle 3,[3]\rangle\oplus & \langle 1,[1,2]\rangle\oplus & \langle 2,[1,2]\rangle\oplus \\
\langle 2,[2,3]\rangle\oplus & \langle 3,[2,3]\rangle\oplus & \langle 1,[1,2,3]\rangle\oplus & \langle 2,[1,2,3]\rangle\oplus & \langle 3,[1,2,3]\rangle\oplus \\
\langle 1,[\,]\rangle\ominus & \langle 1,[2]\rangle\ominus & \langle 1,[3]\rangle\ominus & \langle 1,[2,3]\rangle\ominus & \langle 2,[\,]\rangle\ominus \\
\langle 2,[1]\rangle\ominus & \langle 2,[3]\rangle\ominus & \langle 3,[\,]\rangle\ominus & \langle 3,[1]\rangle\ominus & \langle 3,[2]\rangle\ominus \\
\langle 3,[1,2]\rangle\ominus & & & &
\end{array}
$$

If the literal $\mathsf{components(B,A,C)}$ is now added to the clause body to give

$$\mathsf{member(A,B)\ \text{:-}\ components(B,A,C)}$$

the new clause has three variables and is satisfied by the bindings

$$\langle 1,[1],[\,]\rangle\oplus \quad \langle 2,[2],[\,]\rangle\oplus \quad \langle 3,[3],[\,]\rangle\oplus \quad \langle 1,[1,2],[2]\rangle\oplus \quad \langle 2,[2,3],[3]\rangle\oplus \quad \langle 1,[1,2,3],[2,3]\rangle\oplus$$

For instance, $\langle 1,[1],[\,]\rangle$ is included because the values A=1, B=[1], C=[ ] satisfy the clause body, and is labelled $\oplus$ because the tuple $\langle 1,[1]\rangle$ formed by the values of the clause head variables A and B belongs in member. Since all the bindings are labelled $\oplus$, the clause covers no $\ominus$ tuples and so is complete. The $\oplus$ tuples covered by this clause are removed, leaving only $\langle 2,[1,2]\rangle$, $\langle 3,[2,3]\rangle$, $\langle 2,[1,2,3]\rangle$ and $\langle 3,[1,2,3]\rangle$ to be covered by subsequent clauses in the definition.

The next iteration commences with the remaining $\oplus$ tuples and all $\ominus$ tuples, namely

$$
\begin{array}{rrrrr}
\langle 2,[1,2]\rangle\oplus & \langle 3,[2,3]\rangle\oplus & \langle 2,[1,2,3]\rangle\oplus & \langle 3,[1,2,3]\rangle\oplus & \langle 1,[\,]\rangle\ominus \\
\langle 1,[2]\rangle\ominus & \langle 1,[3]\rangle\ominus & \langle 1,[2,3]\rangle\ominus & \langle 2,[\,]\rangle\ominus & \langle 2,[1]\rangle\ominus \\
\langle 2,[3]\rangle\ominus & \langle 3,[\,]\rangle\ominus & \langle 3,[1]\rangle\ominus & \langle 3,[2]\rangle\ominus & \langle 3,[1,2]\rangle\ominus
\end{array}
$$

If the literal $\mathsf{components(B,C,D)}$ is added to the clause head to give the partial clause

$$\mathsf{member(A,B)\ \text{:-}\ components(B,C,D)}$$

with four variables, the bindings that satisfy this partial clause are

$$
\begin{array}{rrrr}
\langle 2,[1,2],1,[2]\rangle\oplus & \langle 3,[2,3],2,[3]\rangle\oplus & \langle 2,[1,2,3],1,[2,3]\rangle\oplus & \langle 3,[1,2,3],1,[2,3]\rangle\oplus \\
\langle 1,[2],2,[\,]\rangle\ominus & \langle 1,[3],3,[\,]\rangle\ominus & \langle 1,[2,3],2,[3]\rangle\ominus & \langle 2,[1],1,[\,]\rangle\ominus \\
\langle 2,[3],3,[\,]\rangle\ominus & \langle 3,[1],1,[\,]\rangle\ominus & \langle 3,[2],2,[\,]\rangle\ominus & \langle 3,[1,2],1,[2]\rangle\ominus
\end{array}
$$

Adding a further literal to give the new partial clause

$$\mathsf{member(A,B)\ \text{:-}\ components(B,C,D),\ member(A,D)}$$

restricts the bindings to just

⟨2,[1,2],1,[2]⟩⊕ ⟨3,[2,3],2,[3]⟩⊕ ⟨2,[1,2,3],1,[2,3]⟩⊕ ⟨3,[1,2,3],1,[2,3]⟩⊕

For instance, the binding ⟨1,[2],2,[ ]⟩ is now excluded because the values A=1, B=[2], C=2, D=[ ] do not satisfy the requirement that A is a member of D. All bindings are labelled ⊕, again signalling completion of the clause. Each tuple in the target relation is now covered by the clauses

```
member(A,B) :- components(B,A,C).
member(A,B) :- components(B,C,D), member(A,D).
```

so the definition of member is complete. Using the Prolog notation for lists, these clauses might be written

```
member(A,[A|C]).
member(A,[C|D]) :- member(A,D).
```

This example begs some important questions such as how to find appropriate literals to add to the clause body. The next several subsections take up issues of this kind that are central to FOIL's learning method.

2.2 Selecting literals

Literals that can appear in the body of a clause are restricted by the requirement that programs be function-free, other than for constants appearing in equalities. The possible forms that FOIL considers are:

- $Q(V_1, V_2, ..., V_k)$ and $not(Q(V_1, V_2, ..., V_k))$, where Q is a relation and the V_i's denote *existing* variables bound earlier in the clause or *new* variables.
- $V_i{=}V_j$ or $V_i{\neq}V_j$, for existing variables V_i and V_j of the same type.
- $V_i{=}c$ and $V_i{\neq}c$, where V_i is an existing variable and c is a constant of the appropriate type. Only constants that have been designated as suitable to appear in a theory are considered – a reasonable theory for member might reference the null list [ ] but should not involve an arbitrary list such as [1,2].
- $V_i \leq V_j$, $V_i > V_j$, $V_i \leq t$, and $V_i > t$, where V_i and V_j are existing variables with numeric values and t is a threshold chosen by FOIL.

If the learned theory must be pure Prolog, negated literal forms $not(Q(...))$ and $V_i{\neq}...$ can be excluded by an option.

Literals of the forms $Q(...)$ and $not(Q(...))$ are further constrained. At least one variable must have been bound earlier in the partial clause, either by the head or a literal in the body. As with GOLEM, the *depth* of new variables is limited, where variables appearing in the head have depth 0 and a new variable in a literal has depth one greater than the maximum depth of its existing variables. Finally, if Q is the target relation, recursive body literals that could cause non-termination are excluded as discussed in Section 2.3.

A literal in the body of a clause can serve two purposes. It may increase the proportion of $\oplus$ bindings, thereby moving the clause closer to completion when all bindings are $\oplus$. Alternatively, a literal of the form $Q(...)$ may introduce new variables needed in the final clause. Literals of the first kind, referred to as *gainful*, may also introduce new variables, but this is the primary motivation for the second class of *determinate* literals.

Gainful literals are evaluated using an information heuristic. Let the number of $\oplus$ and $\ominus$ bindings of a partial clause be $n^{\oplus}$ and $n^{\ominus}$ respectively. The average information provided by the discovery that one of the bindings has label $\oplus$ is

$$I(n^{\oplus}, n^{\ominus}) = -\log_2 \; n^{\oplus} / \; (n^{\oplus} + n^{\ominus}) \text{ bits.}$$

If a literal L is added, some of these bindings may be excluded and each of the rest will give rise to one or more bindings for the new partial clause. Suppose that k of the $n^{\oplus}$ bindings are not excluded by L, and that the numbers of bindings of the new partial clause are $m^{\oplus}$ and $m^{\ominus}$ respectively. The total information gained by adding L is then

$$k \times (I(n^{\oplus}, n^{\ominus}) - I(m^{\oplus}, m^{\ominus})) \text{ bits.}$$

In the member example, there are 10 $\oplus$ and 11 $\ominus$ bindings at the start of the first clause. Adding components(B,A,C) excludes all but

⟨1,[1]⟩$\oplus$ ⟨2,[2]⟩$\oplus$ ⟨3,[3]⟩$\oplus$ ⟨1,[1,2]⟩$\oplus$ ⟨2,[2,3]⟩$\oplus$ ⟨1,[1,2,3]⟩$\oplus$

each of which gives rise to a single binding for the new clause. The total information gained by adding this literal is then $6 \times (I(10, 11) - I(6, 0))$ or 6.42 bits.

Determinate literals are inspired by GOLEM's *determinate terms* but, whereas GOLEM can learn only theories in which all terms are determinate, FOIL implements the idea as a preference rather than a requirement. A determinate literal is one that introduces new variables such that the new partial clause has exactly one binding for each $\oplus$ binding in the current clause, and at most one binding for each $\ominus$ binding. Determinate literals are useful because they introduce new variables, but neither reduce the potential coverage of the clause nor expand the set of bindings. This is exemplified by the first literal components(B,C,D) of the second clause above; every binding other than those of the form ⟨...,[ ]⟩$\ominus$ yields a single new binding in which new variables C and D are the head and tail of B respectively. Notice that this literal is also gainful as it increases the proportion of $\oplus$ bindings.

All sensible literals derived from all relations are considered when adding literals to a clause. Some literals can be omitted, for instance

- literals that do not satisfy the argument type constraints;
- a literal $Q(..., X, ..., X, ...)$ with the same variable in argument positions i and j, when no tuple in the relation Q has the same constant in positions i and j; and
- recursive literals that might cause infinite recursion (see below).

Further, evaluation of a literal can often be abandoned when it becomes clear that it is not determinate and cannot come close to the gain of the most gainful literal found so far. On occasion a literal can be omitted altogether from consideration because it is a specialisation of a literal already known to exclude too many $\oplus$ bindings.

2.3 Assuring recursive soundness

Theories found by FOIL are intended to be executable as Prolog programs, so it is important that recursive theories do not lead to infinite recursion. To this end, FOIL incorporates a sophisticated scheme that bars recursive literals unless they can be proven to be problem-free, at least to the extent of ensuring termination on ground queries to a single target relation[1]. The approach, described in detail in [Cameron-Jones and Quinlan, 1993a], has three phases:

Ordering constants: The constants of each type T can be given to FOIL in their natural order, if one exists, or FOIL can find a plausible ordering. In the latter case, each pair of arguments A_i, A_j of type T in every relation R is examined to see whether the tuples of constants defining R are consistent with a partial order, here denoted $A_i \prec A_j$ (since it is impossible to distinguish between $A_i < A_j$ and $A_i > A_j$). If the arguments exhibit such a partial order, each tuple in relation R will give $c_i \prec c_j$ for the constants c_i, c_j in the ith and jth positions respectively. The argument partial order is ruled out only when the closure of these inequalities between constants implies $c_k \prec c_k$ for some constant c_k.

Having found all potential partial orderings of pairs of arguments of type T across all relations, FOIL orders the constants of type T to be consistent with the maximum number of the argument partial orders. This process is carried out just once for each type for which an ordering is not specified by the user.

Ordering pairs of variables: The ordering of constants of type T may imply an ordering of pairs of variables V_i, V_j of type T in a partial clause. Each binding of the partial clause specifies values c_i, c_j for V_i and V_j; if it is always the case that $c_i \prec c_j$, then $V_i \prec V_j$.

Ordering recursive literals: Recursive termination will be assured if, for all clauses with head $R(V_1, V_2, ...)$ and body literal $R(W_1, W_2, ...)$, the body literal is less than the head. To order literals, FOIL considers schemes of the form

[1] That is, with no mutually recursive definitions of two or more relations.

$$\begin{aligned}&R(W_1, W_2, ...) < R(V_1, V_2, ...) \text{ if}\\ &\quad W_\alpha <_\alpha V_\alpha, \text{ or}\\ &\quad W_\alpha = V_\alpha \text{ and } W_\beta <_\beta V_\beta, \text{ or}\\ &\quad W_\alpha = V_\alpha \text{ and } W_\beta = V_\beta \text{ and } W_\gamma <_\gamma V_\gamma, \text{ or ...}\end{aligned}$$

for suitable argument positions α, β, γ, ... and where $<_i$ denotes $<$ if the real order of constants is known, otherwise a choice between $\prec$ or $\succ$. Whenever a recursive literal is being considered, FOIL tries to construct a literal ordering scheme of this kind that is satisfactory for both this literal and all other recursive literals in the definition so far. If such a scheme does not exist, the recursive literal is ruled out.

For the member example, FOIL finds that the definition of the components(L,H,T) relation is consistent with T$\prec$ L and orders the list constants

$$[\,] \prec [1] \prec [2] \prec [3] \prec [1,2] \prec [2,3] \prec [1,2,3].$$

Now consider the partial clause

member(A,B) :- components(B,C,D)

where, by the ordering above, D$\prec$ B. When considering the addition of the recursive literal member(A,D), it is clear that an ordering scheme

member(W_1, W_2) $<$ member(V_1, V_2) iff $W_2 \prec V_2$

will guarantee that the body literal is less than the head and so ensure that this literal cannot cause infinite recursion.

Many first-order learning systems employ simpler mechanisms[2] to prevent problematic recursion, or no mechanisms at all! Even though this scheme is relatively complex, it is computationally efficient in practice and is necessary for learning more difficult recursive definitions such as Ackermann's function (discussed in Section 3.2).

2.4 Controlling search

FOIL's exploration of the space of possible definitions is fundamentally greedy, but the system incorporates mechanisms to curtail search down a particular path and to recover from poor choices of literals. Recovery is achieved by establishing *checkpoints* when a gainful literal added to a clause is only marginally better than an alternative literal. A small, fixed number of checkpoints (default 20) is maintained and, if the current partial clause cannot be completed so as to exclude all $\ominus$ tuples, search is restarted from the best remaining checkpoint. This non-chronological backtracking is invoked relatively infrequently since greedy search is usually sufficient to find a clause. Backtracking is not used to attempt to find a *better* clause, although this could become an option in future versions.

---

[2]Early versions of FOIL used a weaker scheme that required one argument of the body literal to be less than the corresponding argument of the head.

Greedy search can fail either because there is no literal that could be added to a clause or, more commonly, because the addition of another literal will render the clause too complex with respect to the training data. The complexity criterion is based on Rissanen's Minimum Description Length Principle [Quinlan and Rivest, 1989] and requires that the cost of encoding a clause should never exceed the cost of identifying explicitly the tuples that it covers. Since determinate literals are added indiscriminately, they are excluded from the calculation of the cost to encode a clause, defined as the number of bits needed to identify the relation and arguments of all non-determinate literals in the clause body. The cost of identifying the n tuples that it covers among the $\oplus$ and $\ominus$ tuples of the target relation is the logarithm to base 2 of the number of ways in which n tuples could be selected. This criterion thus rules out elaborate clauses that cover few tuples.

When exploring literals to add to the developing clause body, FOIL sometimes notices a literal that would complete the clause but prefers another literal that is determinate or has higher gain. The best of the complete clauses encountered during search is retained in the wings and, if the final clause is not superior in terms of compactness or coverage, the saved clause is substituted in its place.

The final modification to straightforward search occurs when a literal L chosen for addition to the partial clause contains only variables that appear in the clause head. L could have appeared as the first literal of the clause body while intervening literals introducing new variables might have restricted the clause's coverage. In such situations, all non-determinate literals that introduce variables are discarded and search resumes from the shortened partial clause.

2.5 Pruning definitions

A particular literal in a completed clause may be needed because it prevents the clause from covering $\ominus$ tuples, because it introduces a variable used in a later literal, or because it establishes a partial order on which recursion control depends. As a consequence of its incremental construction, a clause may contain literals that serve none of these purposes. Removal of such literals has two benefits: the clause becomes simpler, and it may also cover more $\oplus$ tuples of the target relation.

The policy of adding all determinate literals to the clause body is the principal source of unnecessary literals. Consequently, clause pruning proceeds in two stages. All determinate literals that introduce variables not used by later literals are removed. This operation is fast but fallible, so the shortened clause is tested to see that it is still valid and recursively sound; if not, the original clause is restored. Then a literal-by-literal pruning process is carried out, starting from the last literal in the clause body. At each step a literal is removed, the residual clause tested, and the literal restored only if the pruned clause is unsatisfactory. This iterative pruning can be costly when the initial clause is long, but generalising the clause as much as possible can expedite learning of the rest of the definition since fewer $\oplus$ tuples remain to be covered.

Definitions themselves can also be redundant since the $\oplus$ tuples covered by early clauses may also be covered by later clauses. When the definition is complete, each clause is examined to see whether it uniquely covers one or more $\oplus$ tuples; if not, the clause is discarded.

2.6 Dealing with closed worlds

We now come to the first of the more recent developments in FOIL. Unlike aspects described above, these are not documented elsewhere and so are presented in more detail.

FOIL requires that the target and background relations be defined extensionally as tuples of constants. This cannot be done when the relation is inherently infinite, so the usual practice is to specify a finite closed world and to limit tuples to those containing only constants that appear in the closed world. This implicitly assumes that a satisfactory definition for the closed world will be correct in general, even when used in conjunction with intensionally-defined background knowledge. Bell and Weber [1993] call this the *open domain* assumption.

Consider the task of learning the concept of a simple list as one that contains at most one element. We might establish a closed world consisting of all lists with up to three elements drawn from {1,2,3} in which simple(L) is defined by the tuples $\{\langle[\,]\rangle, \langle[1]\rangle, \langle[2]\rangle, \langle[3]\rangle\}$. Background relations are components(L,H,T) as before, and conc(A,B,C) meaning that the result of concatenating lists A and B is list C. Notice that conc does not contain tuples representing the result of concatenating two two-element or two three-element lists, since these would form lists that lie outside the closed world.

For this task, FOIL immediately finds the surprising definition

simple(A) :- conc(A,A,B).

The definition is correct for the closed world since, when A has two or more elements, the result of concatenating A with itself lies outside the closed world and the corresponding tuple does not appear in conc. Unfortunately, though, this definition is not correct in general.

Enlarging the closed world merely postpones the problem. In a new closed world including all lists up to length four, for example, FOIL finds a similar definition

simple(A) :- conc(A,A,B), conc(B,B,C).

This is still correct for the larger closed world – if A has two or more elements then B has four or more and so the result of concatenating B to itself again is not defined in the closed world.

This problem is not restricted to FOIL but is a consequence of using extensionally defined relations. For instance, GOLEM [Muggleton and Feng, 1992] also requires relations to be defined by ground assertions and finds identical definitions for these tasks.

The solution we have implemented in FOIL involves a special constant $\wedge$ denoting *out-of-world*. In the three-element world, the definition of conc would include the tuple $\langle$[1,2,3],[1,2,3],$\wedge\rangle$ to indicate that the result of concatenating [1,2,3] to itself is not defined in the closed world. This constant $\wedge$ has special significance for FOIL: a literal is barred if adding it to the clause body would cause $\wedge$ to appear in any of the bindings. The rationale for this is that all definitions are forced to stay within the closed world and cannot exploit boundary effects attributable to its finite size.

Returning to the example, we see that $\ominus$ tuples for simple include $\langle$[1,2,3]$\rangle$. The literal conc(A,A,B) is therefore excluded since it would generate a binding $\langle$[1,2,3],$\wedge\rangle$. The definition now found by FOIL is more complex:

```
simple([]).
simple(A) :- components(A,B,[]).
```

or, in Prolog notation,

```
simple([]).
simple([B]).
```

This definition satisfies the open domain assumption since it is correct in general, not just for the particular closed world in which it was learned.

2.7 Making clauses more understandable

An important goal of all symbolic learning is to find theories that make sense to people. To this end, FOIL contains mechanisms intended to re-express clauses in more intuitive form. Some transformations are relatively easy, such as removing literals V_i=V_j and V_j=c from the body by substituting V_i or c respectively for each occurrence of V_j. For instance, the first clause of the definition above initially has the form

```
simple(A) :- A=[].
```

The body literal was removed and [ ] substituted for A in the head.

Such simple transformations are not sufficient to render some clauses intelligible, even after pruning. An example of this arises while FOIL is learning a definition of sort(A,B) given just the background relations components(L,H,T) and less-than(A,B). After learning the base case (sorting the null list gives itself), FOIL embarks on a second clause. The literals added to the clause body are

components(A,C,D), components(B,E,F)	(both determinate)
sort(D,G)	(determinate)[3]
components(H,C,G)	(determinate)
B=H	(gainful)
D=[ ]	(gainful).

After pruning and substitution, the clause becomes

sort(A,B) :- components(A,C,[ ]), sort([ ],G), components(B,C,G).

which is equivalent to

sort([C],[C|G]) :- sort([ ],G).

This clause is correct – it forces A and B to be identical single-element lists – but it is certainly not intuitive! The fundamental problem is that literals in the clause body establish implicit equalities that must be made explicit if the clause is to be intelligible. For example, sort([ ],G) forces G to be the null list in all bindings, but the literal G=[ ] does not appear in the clause. We have found that addition of such implicit literals to the clause before pruning often leads to a simpler clause.

When a clause is completed, its variable bindings are examined for equalities of the form V_i=V_j or V_j=c that do not appear explicitly in the clause body. Any such equalities are inserted into the clause immediately after the first literal that binds V_j. Explicit equalities are also promoted within the clause, the goal being to retain them in the pruned clause as long as possible. The clause is then pruned from the end in the usual way.

In the case of this clause, the implicit equalities established by the literals are A=B, A=H, C=E, F=[ ] and G=[ ]. When these are inserted and the literal D=[ ] promoted, the clause body becomes

A=B,
components(A,C,D),
D=[ ],
components(B,E,F),
C=E,
F=[ ],
sort(D,G),
G=[ ]
components(H,C,G),
A=H,
B=H.

[3]It might seem that there should be a corresponding determinate literal sort(F,H). However, F is the tail of a sorted list and is therefore sorted already; thus H=F and this literal introduces no new variables.

All but the first three literals are now pruned and, after substitution, this base case clause becomes much more recognisable as

```
sort(A,A) :- components(A,C,[ ]).
```

or

```
sort([C],[C]).
```

3. Applications

This section examines a representative sample of tasks to which FOIL has been applied. Our intention is to demonstrate that the system's approach is effective across a range of domains encompassing most areas of first-order learning.

The original FOIL paper [Quinlan, 1990] presents results on six families of tasks addressed by other learning systems, including classics such as the definition of an arch [Winston, 1975], classifying trains [Michalski, 1980], discovering rules for the card game Eleusis [Dietterich, 1980], and deciding when chess positions are illegal [Muggleton *et al*, 1989]. Several experiments involving larger datasets or more difficult definitions have subsequently been completed; two are reported here for the first time.

3.1 Recursive list-processing functions

Perhaps our most comprehensive study comes from the domain of learning simple list-processing functions, reported in [Quinlan and Cameron-Jones, 1993]. All the 18 such tasks presented in Chapter 3 of Bratko's [1990] well-known Prolog text are tackled by FOIL. Two closed worlds are defined, containing respectively all lists of length up to three using elements {1,2,3} and all lists of length up to four using elements {1,2,3,4}. For each function, the target relation is specified exhaustively over the particular closed world so that there is no question of the system's performance being influenced by the choice of examples. The background relations include components and all functions that appear in the previous tasks, most of which are irrelevant to the task at hand.

In almost all cases FOIL is able to find a satisfactory definition, although some definitions are correct only in the closed world.[4] In one case, FOIL finds a more concise definition than that given in the book! The relation dividelist(A,B,C) is intended to put alternate elements from A into lists B and C. Bratko gives a three-clause definition, whereas FOIL's has just two clauses:

[4] Later versions of FOIL, especially since the introduction of the out-of-world constant $\wedge$, overcome most of the remaining problems on these tasks.

```
dividelist([ ],[ ],[ ]).
dividelist(A,B,C) :- components(A,D,E), components(B,D,F),
                     dividelist(E,C,F).
```

where the second clause might be written

```
dividelist([D|E],[D|F],C) :- dividelist(E,C,F).
```

Other list-processing functions have been investigated, notably learning the quicksort procedure [Quinlan, 1991].

3.2 Arithmetic functions

Functions such as n-choose-m can also be learned from small closed worlds. The most complex studied to date is Ackermann's function, defined as

$$F(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ F(m-1,1) & \text{if } n = 0 \\ F(m-1, F(m,n-1)) & \text{otherwise} \end{cases}$$

In function-free form, the corresponding predicate Ackermann(A,B,C) means $F(A,B) = C$. From a closed world containing integers 0 to 20 and a background relation succ(A,B) meaning $B = A+1$, FOIL takes 16.7 seconds on a DECstation 3000/800 to find the definition

```
Ackermann(0,B,C) :- succ(B,C).
Ackermann(A,0,C) :- succ(D,A), Ackermann(D,1,C).
Ackermann(A,B,C) :- succ(D,A), succ(E,B), Ackermann(A,E,F),
                    Ackermann(D,F,C).
```

This program is interesting because it contains two recursive clauses, one being doubly recursive. Learning this last clause requires subtle control of recursion since the literal Ackermann(A,E,F) decreases the second argument while Ackermann(D,F,C) increases the second argument but decreases the first. FOIL is the only system we know of that is capable of learning this definition.

3.3 Attribute-value data

Since the theory language available to FOIL encompasses all symbolic zeroth-order theories, it is relevant to enquire how the performance of FOIL compares to that of zeroth-order systems on attribute-value tasks. A group of two-class classification tasks was investigated, using no background relations and target relations of the form Class1(V_1,V_2,...) and Class2(V_1,V_2,...) with one argument for each attribute. Experiments were carried out, first restricting FOIL to literals of the forms V_i=c, V_i>t and $V_i \leq t$ (giving exactly the same theory language available to most zeroth-order learning systems), then allowing an extended language including literals such as V_i=V_j and V_i>V_j that compare the values of pairs of attributes.

Results of these experiments appear in [Cameron-Jones and Quinlan, 1993b]. Our general conclusion is that FOIL performs slightly better than C4.5 [Quinlan, 1993] on these datasets, especially when permitted to use the extended theory language, but that learning generally requires more computation. The theories found by FOIL are often less simple than those found by C4.5, indicating that the mechanism to limit clause complexity described in Section 2.4 does not adequately prevent overfitting of the training data. This finding is supported by other researchers such as Fürnkranz [1993].

3.4 Protein secondary structure

More evidence for overfitting comes from another task of learning to predict protein secondary structure [Muggleton, King and Sternberg, 1992]. Proteins consist of long chains of amino acid residues and at certain positions they form structures such as α-helices and β-sheets. The target relation here is alpha(Protein,Position) that indicates when an α-helix occurs at the specified position in a particular protein. Twenty-five background relations identify the residue at each position and provide chemical and physical properties of the residues. The training set consists of 1612 tuples taken from twelve proteins with a further 416 tuples from four different proteins used as a test set.

GOLEM, augmented with a hand-crafted criterion to avoid overfitting in this domain, is able to find 21 clauses that exhibit a predictive accuracy of 72% on the test set. FOIL performs relatively poorly, finding 84 clauses that have an accuracy of 65% on the test tuples, 7% lower than the corresponding figure for GOLEM.

[GOLEM's performance on this task is further improved by a form of bootstrapping. The first (level 0) theory learned above predicts occurrences of α-helices additional to those recorded in the training data. When these are added as new $\oplus$ tuples, GOLEM learns a revised (level 1) theory from the modified data. Repeating the process gives a level 2 theory whose accuracy on the test data jumps to 81%.]

3.5 Identifying document components

We come now to the first new application reported in this paper – learning rules to locate the logical components of a document such as that shown in Figure 2. Different documents have varying numbers of components and relationships (such as alignment) between pairs of components, so this is a good example of a task that is ill-suited to zeroth order learning methods based on fixed-length attribute-value vectors.

Five target relations identify document components relevant to sender, receiver, date, reference and logo. Plentiful background information is provided by 57 relations that describe 244 components of 20 single-page documents, giving each component's size, type (e.g. text, picture), position on the page, and alignment with other components. The document x1 of Figure 2 with ten components x2...x11 is described by 102 tuples

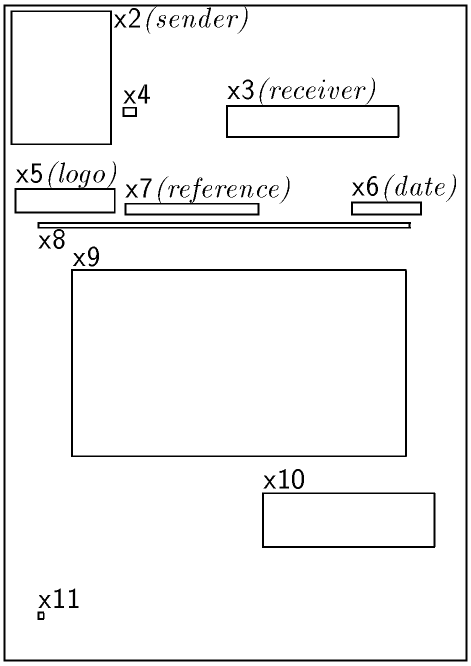

Figure 2: Sample document showing components (following Semeraro *et al* [1993]).

in the background relations.

Results of a leave-one-out cross-validation appear in Table 1. In each run, information about components of one document is omitted from the training data and used to test the theory learned from the components of the remaining documents, the same procedure being repeated for each target relation and each document. Test errors are broken down into false positives ($\ominus$ tuples incorrectly predicted to belong to the target relation) and false negatives ($\oplus$ tuples not covered by the learned theory). Accuracy on unseen test data is excellent, ranging from 100% for sender and logo to 96.3% for date.

Table 1: Results on unseen data, document identification tasks.

Target Relation	False Pos	False Neg	Total Errors	Error Rate
sender	0	0	0	–
receiver	3	5	8	3.3%
date	5	4	9	3.7%
reference	2	2	4	1.6%
logo	0	0	0	–

Table 2: Results on chess endgame

Moves	Positions	FOIL Clauses	FOIL Uncovered	FOIL Total	GCWS Clauses
zero	27	3	0	3	6
one	78	5	8	13	12
two	246	15	0	15	20
three	81	15	2	17	44
four	198	21	20	41	89
five	471	50	5	55	185
six	592	61	21	82	–
seven	683	90	24	114	–
eight	1433	93	87	180	–
nine	1712	128	91	219	–
ten	1985	153	95	248	–
eleven	2854	180	165	345	–
twelve	3597	148	352	500	–
thirteen	4194	224	102	326	–
fourteen	4553	200	31	231	–
fifteen	2166	68	0	68	–
sixteen	390	3	0	3	–
(drawn)	2796				

3.6 Moves to win in a chess endgame

The final application concerns the simplest chess endgame, King and Rook versus King. Bain [1994] studies the task of learning to predict the minimum number of moves required for a win by the Rook's side (with values 0 through 16) or, failing this, a draw – there are no positions in which the Rook's side loses.

Bain formulates this problem as a cascade of learning tasks. From a database of all legal positions after the removal of symmetric variants, a theory is learned that describes positions won in zero moves. These positions are then eliminated from the data and the next task, discriminating positions won in one move from drawn positions and those won in two or more moves, is presented to the learning system. The process continues in a similar fashion with the final theory discriminating positions won in 16 moves from drawn positions. Bain uses a system called GCWS, based on GOLEM, that allows exception predicates to be invented and used in clauses; with it, he finds correct definitions for the first six levels of this task.

Table 2 summarises results obtained when the experiment was repeated using FOIL rather than GCWS. For each number of moves to win, the Table shows the number of $\oplus$ tuples that must be covered by the learned theory. The definitions found by FOIL often fail to cover all $\oplus$ tuples so, following the practice used by GOLEM, uncovered $\oplus$ tuples are added as ground clauses; the number of clauses in the final theory appears in the column labelled Total. For comparison, the final column shows the size of the theories constructed by GCWS.

FOIL handles this domain comparatively well. Correct definitions are found for each number of moves, with one exception – the definition of positions lost in eleven moves has one false positive error. FOIL's definitions generally compress the data more than those found by GCWS, with some exact clauses being remarkably simple. Even better results are obtained if `drawn` positions are identified first, then positions lost in one move and so on, leaving the final theory to distinguish between positions lost in fifteen and sixteen moves.

4. Related Systems

Elements of FOIL's approach have been used in other systems, often with considerable modification and innovative extension. These developments are typically aimed at broadening the learning task itself (such as by taking account of additional domain knowledge), correcting some perceived deficiency in FOIL (such as its tendency to overfit), or specialising it for a particular family of tasks (such as learning control heuristics).

FOCL [Pazzani, Brunk and Silverstein, 1991; Pazzani and Kibler, 1992] is an early extension of FOIL that takes advantage of domain knowledge in the form of a partial theory, intensionally-specified background relations, and relational clichés. The prior theory may contain clauses that are too general in that they cover $\ominus$ tuples, and too specific in failing to cover $\oplus$ tuples of the target relation. To investigate this, the prior theory is elaborated by unfolding its proof tree, guided by the same information metric that FOIL uses to select literals to be added to clauses, and complete paths in the tree that remain too general are specialised by invoking FOIL's literal-adding procedure. Background relations defined as clauses rather than as sets of tuples are evaluated intensionally and, when a clause of a background definition has high gain, an appropriate specialisation of the clause body is added to the current partial clause. Similarly, relational clichés consist of schemas containing sequences of literals that tend to belong together in definitions; FOIL may miss such combinations unless at least one of the individual literals is determinate or has high gain. FOCL thus represents a clean union of ideas from explanation-based learning and empirical induction.

Another system from UCI, AUDREY II [Wogulis and Pazzani, 1993], uses similar mechanisms to specialise over-general theories and to add new clauses, both within a theory-revision context. Rather than being limited to adding literals, however, this system uses four revision operators that include replacing some literals in an existing clause.

Several other researchers have modified FOIL to make it more robust, especially with respect to noisy data. mFOIL [Lavrač and Džeroski, 1994] replaces FOIL's greedy search with beam search, thereby increasing the chances of finding a good clause; chooses literals to add to the clause body on the basis of the estimated accuracy of the new clause, rather than on information gain; and uses a statistical significance test instead of the MDL criterion to decided when a clause should not be allowed to grow further. FOSSIL [Fürnkranz, 1993] employs a single correlation criterion both for selecting the next literal to add and for stopping the growth of a clause. These systems perform much better than FOIL on a chess-derived relation illegal [Muggleton *et al*, 1989] corrupted by moderate levels of noise, learning more compact definitions with higher predictive accuracy.

HYDRA [Ali and Pazzani, 1993] deals with noise by extending FOIL in three dimensions. The learning task is widened to allow for any number of classes rather than just the definition of a (binary) target relation. HYDRA then constructs a definition for each class; in our context, this involves learning separate definitions for the target relation R and for *not(R)*. Since the language of clauses is not closed under negation, one definition might be considerably simpler and more robust than the other. Secondly, the reliabilities of individual clauses in all theories are estimated from likelihood ratios derived from their respective coverages of $\oplus$ and $\ominus$ tuples. A query is evaluated against all theories, e.g. against both the theory for R and the theory for *not(R)*. The outcome is determined by the most reliable clause from any theory that succeeds on the query. Finally, HYDRA uses likelihood improvement rather than information gain to select the next literal to be added to the clause body. Ablation experiments suggest that all three changes help to produce more robust learning.

A most promising area for relational learning is the formulation of control heuristics. DOLPHIN [Zelle and Mooney, 1993] blends ideas from explanation-based learning and induction with the goal of making logic programs run faster. The central idea is to insert a guard literal useful-R-k($query$) as the first body literal in each nondeterministic clause k of relation R, preventing the clause from being evaluated unless it is judged likely to succeed. The $\oplus$ and $\ominus$ tuples of this relation are provided by examples of when the particular clause succeeds and fails that are extracted from an execution trace of the original program; from these, a learning program finds a definition of the guard literal. Although the learning program is based on FOIL, it embodies an innovative method of specialising incomplete clauses. The proof of the original query is generalised by replacing constants with unique variables and from it DOLPHIN constructs a set of *specialisation pairs* $\langle G, L\rangle$, where G is a solved subgoal and L is either *true* or an operational literal from the proof that shares one or more variables with G. Each such pair provides a candidate specialisation of a partial clause H :- B obtained by unifying head H with G (with most general unifier σ), and altering the clause to $\sigma(H$:- $B, L)$. This allows the head of a clause, as well as its body, to be specialised and considers only new body literals that are known to be relevant to part of the proof. FOIL's information gain criterion is then used to select a specialisation from the candidates above. In one impressive example, DOLPHIN is able to transform a naive permute-and-check sorting algorithm of complexity $O(n!)$ to an $O(n^2)$ insertion sort.

The same authors have developed another similarly-motivated system Chillin [Zelle, Mooney and Konvisser, 1994; Zelle and Mooney, 1994] that has learned search control rules for a nondeterministic English parser. The initial theory consists of ground clauses obtained directly from the $\oplus$ tuples of the target relation. Successive steps compress this definition by introducing more general clauses and removing subsumed clauses. A more general clause is found by selecting two existing clauses, forming the head of a new clause as the least general generalisation of their heads in the manner of GOLEM, then specialising the clause by adding literals to the clause body. This last stage is similar to FOIL, except that the metric used to select literals is based on notions of compression rather than information gain. Chillin also includes a mechanism derived from CHAMP [Kijsirikul, Numao and Shimura, 1992] that can assess the benefit of introducing a new relation and learning its definition.

Grasshopper [Leckie and Zukerman, 1993] is another interesting system that learns to control search in planning domains. Examples of search decisions, both good and bad, are extracted from the planner's execution trace and grouped according to the planning goal that they address and the action chosen. A learning algorithm based on FOIL generalises the examples of each group to produce search control heuristics. In a final step, the utility of the learned rules is assessed by comparing their evaluation cost against their benefit in reduced search cost, leading to an optimised subset that minimises overall planning time.

The overview of FOIL presented in Section 2.1 talks only of learning a definition for a single target relation. The implementation, however, allows for any number of target relations; FOIL simply tackles them one after another. De Raedt, Lavrač and Džeroski [1993] point out that there are situations in which mutually recursive target relations can lead to non-terminating programs – recall the caveats to recursive soundness in Section 2.3. To overcome this problem, their system MPL develops all definitions of target relations in parallel, checking for global as well as local consistency and using heuristics for specialising partial clauses that are similar to mFOIL's.

A quite different kind of extension is embodied in Grendel and Grendel2 [Cohen, 1994a,b]. Relations with high arity can pose severe computational problems for FOIL since, if there are v variables in a partial clause, a relation of arity r can give rise to $O((v+r)^r)$ potential next literals. Even when many or most of these are ruled out by type constraints and the like, the remaining candidates might still be too numerous to contemplate. Further, high-arity relations may require an impossibly large set of $\ominus$ tuples if over-generalised clauses are to be avoided. Grendel2 attacks this problem within a FOIL-like framework by specifying a *hypothesis language* that restricts the form of definitions to those that make sense in the domain. This not only prevents consideration of useless literals and literal combinations, but can also serve in place of $\ominus$ tuples to prevent over-generalisation; the goal is then to find a definition in the hypothesis language that covers the $\oplus$ tuples of the target relation. Cohen [1994a] discusses an application to reverse engineering in which the goal is to reconstruct the specification of a database interface consisting of over a million lines of C. Grendel2 is able to recover an accurate description of one-third of the system, despite the presence of relations with high arity and clause-level domain constraints that defeat FOIL.

5. Areas for Further Research

Systems such as the above extend the basic general-to-specific paradigm for inducing first-order theories. The issues that they address are important for the development of more powerful and flexible learning methods, and many more issues remain to be tackled in this vigorous research area. In this Section we raise a couple of fundamental problems that limit FOIL and that, we suspect, apply in some degree to most first-order systems.

5.1 Irrelevant information

Any learning problem can be made harder by adding unhelpful information. The effect is to increase the space of possible theories that could be learned, thereby enlarging the haystack in which we are searching for a figurative needle. In zeroth-order systems, where this problem is synonymous with the presence of irrelevant attributes, effective methods for weeding out the non-useful features have been developed [e.g. John, Kohavi and Pfleger, 1994; Moore and Lee, 1994]; in a sense, the problem is under control. In first-order learning, on the other hand, irrelevant information in the form of unnecessary relations and/or useless fields of relations can have a dramatic impact on learning time.

An example comes from the list-processing tasks discussed in Section 3.1 using the smaller closed world of three-element lists. The first task is to learn a definition of member and FOIL requires only 0.03 seconds to find the definition of Section 2.1. If the second relation conc(A,B,C) is included as an additional background relation, the time required to learn the same definition jumps to 1.36 seconds, or more than 40 times as long. Similarly, adding this excess relation increases GOLEM's learning time by a factor of 60, although it now learns a different definition

member(A,[B|C]) :- conc(D,[A|E],[B|C]).

The impact of extra relations is somewhat unpredictable. Although learning a definition of dividelist, the last task in the original series, does not make use of any of the 14 relations defined by preceding tasks, deleting them produces a comparatively small reduction in FOIL's learning time from 42 seconds to 16 seconds.

Practical learning systems will need to be able to deal with large volumes of information, selecting only that part relevant to the task at hand. We regard this as the most pressing unsolved problem in first-order learning.

5.2 Incomplete information

When learning recursive definitions, most first-order systems require that the set of $\oplus$ tuples for the target relation be largely complete. The few exceptions constrain the form that definitions can take, or depend on information additional to the tuples themselves. FORCE2 [Cohen, 1993] limits definitions to two clauses, one base case and one linearly recursive clause, and requires that instances of the base clause be identified. CRUSTACEAN [Aha, Lapointe, Ling and Matwin, 1994] searches for definitions consisting of a unit base clause and a single recursive clause containing one literal in its body. Both systems can then learn accurate definitions from sparse, random samples of tuples from the target relation. Although restricted theory languages such as these are adequate for a surprisingly large class of relations, there does not seem to be an easy way to extend approaches of this kind towards more complex recursive definitions.

From FOIL's perspective, the problem is that the utility of a clause

$$R(V_1, V_2, ...) \text{ :- } ...., R(W_1, W_2, ...), ...$$

may not become apparent unless there are numerous ground instances of the clause in which the ground instances of $\langle V_1, V_2, ...\rangle$ and $\langle W_1, W_2, ...\rangle$ both belong to R; without this, the recursive literal $R(W_1, W_2, ...)$ has low gain. Even when there are relatively few missing $\oplus$ tuples, FOIL may propose additional clauses to cover what seem to be special cases.

Learning a definition of member again illustrates this. When 10 of the 75 $\oplus$ tuples of member are deleted at random, FOIL finds the definition

```
member(A,B) :- components(B,A,C).
member(A,B) :- components(B,C,D), components(D,A,E).
member(A,B) :- components(B,C,D), member(A,D).
```

or

```
member(A,[A|C]).
member(A,[C,A|E]).
member(A,[C|D]) :- member(A,D).
```

Notice that the second clause has been added to cover "exceptions" to the general rule given by the first and third clause. From the same data, GOLEM learns a similarly verbose definition

```
member(A,[A|B]).
member(A,[B,A|C]).
member(A,[B,C|D]) :- member(A,[B|D]).
```

It might seem as though this problem can be solved simply by evaluating clauses intensionally when removing the $\oplus$ tuples that they cover. However, FOIL often learns a recursive clause before finding a base case, the latter being essential for any intensional coverage at all. A better approach might use the same kind of bootstrapping employed by Muggleton *et al* with the protein data (Section 3.4). At each iteration, covered tuples that do not appear explicitly in either the $\oplus$ or $\ominus$ tuples would be added to the former. In this way it may be possible to assemble a more complete extensional specification of the target relation, leading to a more accurate definition.

6. Conclusion

After some five years of development, FOIL has reached a kind of adolescence: it exhibits some interesting behaviours but has not yet matured sufficiently to withstand the crucible of large real-world applications. Several extensions of its basic approach show great promise, especially in areas like learning control heuristics. We are confident that further research on general-to-specific induction over the next few years will lead to powerful tools for learning in first-order domains.

The current version of FOIL (written in C) is available by anonymous ftp from ftp.cs.su.oz.au, directory pub, file foil6.sh.

Acknowledgements

This research was made possible by a grant from the Australian Research Council and assisted by research agreements with Digital Equipment Corporation. We thank Stephen Muggleton, Giovanni Semeraro and Michael Bain for providing the protein, document and KRK datasets respectively. We are grateful to William Cohen and Stephen Muggleton for most helpful comments on a draft of this paper.

References

1. Aha, D.W., Lapointe, S., Ling, C.X., and Matwin, S. (1994). Learning recursive relations with randomly-selected small training sets. *Proceedings Eleventh International Conference on Machine Learning*, New Brunswick, New Jersey, 12-18. San Francisco: Morgan Kaufmann.

2. Ali, K., and Pazzani, M.J. (1993). HYDRA: a noise-tolerant relational concept learning algorithm. *Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1064-1070. San Francisco: Morgan Kaufmann.

3. Bain, M.E. (1994). Learning logical exceptions in chess. PhD thesis, Department of Statistics and Modelling Science, University of Strathclyde, Scotland.

4. Bell, S., and Weber, S. (1993). On the close logical relationship between FOIL and the frameworks of Helft and Plotkin. *Proceedings Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, 127-147.

5. Bratko, I. (1990). *Prolog Programming for Artificial Intelligence* (2nd edition). Wokingham, UK: Addison-Wesley.

6. Cameron-Jones, R.M., and Quinlan, J.R. (1993a). Avoiding pitfalls when learning recursive theories. *Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1050-1057. San Francisco: Morgan Kaufmann.

7. Cameron-Jones, R.M., and Quinlan, J.R. (1993b). First order learning, zeroth order data. *Proceedings AI'93 Australian Joint Conference on Artificial Intelligence*, Melbourne, 316-321. Singapore: World Scientific.

8. Cameron-Jones, R.M., and Quinlan, J.R. (1994). Efficient top-down induction of logic programs. *SIGART*, 5, 33-42.

9. Cohen, W.W. (1993). Pac-learning a restricted class of recursive logic programs. *Proceedings Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, 73-86.

10. Cohen, W.W. (1994a). Recovering software specifications with inductive logic programming. *Proceedings AAAI-94 Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, 142-148. Menlo Park: AAAI Press.

11. Cohen, W.W. (1994b). Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence, 68*, 303-366.

12. De Raedt, L., Lavrač, N., and Džeroski, S. (1993). Multiple predicate learning. *Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1037-1042. San Francisco: Morgan Kaufmann.

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

14. Dietterich, T.G. (1980). The methodology of knowledge layers for inducing descriptions of sequentially ordered events. Technical Report R-80-1024, Department of Computer Science, University of Illinois at Urbana-Champaign, USA.

15. Fürnkranz, J. (1993). FOSSIL: a robust relational learner. Technical Report TR-93-28, Austrian Research Institute for Artificial Intelligence, Vienna.

16. Gold, E.M. (1967). Language identification in the limit. *Information and Control, 10*, 447-474.

17. John, G.S., Kohavi, R., and Pfleger, K. (1994). Irrelevant features and the subset selection problem. *Proceedings Eleventh International Conference on Machine Learning*, New Brunswick, New Jersey, 121-129. San Francisco: Morgan Kaufmann.

18. Kijsirikul, B., Numao, M., and Shimura, M. (1992). Discrimination-based constructive induction of logic programs. *Proceedings AAAI-92 Tenth National Conference on Artificial Intelligence*, San Jose, CA, 44-49. Menlo Park: AAAI Press.

19. Lavrač, N., and Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications.* London: Ellis Horwood.

20. Leckie, C., and Zukerman, I. (1993). An inductive approach to learning search control rules for planning. *Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1100-1105. San Francisco: Morgan Kaufmann.

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

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

23. Moore, A.W., and Lee, M.S. (1994). Efficient algorithms for minimizing cross-validation error. *Proceedings Eleventh International Conference on Machine Learning*, New Brunswick, New Jersey, 190-198. San Francisco: Morgan Kaufmann.

24. Muggleton, S., and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings Fifth International Conference Machine Learning*, Ann Arbor, Michigan, 339-352. San Mateo: Morgan Kaufmann.

25. 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* Ithaca, NY. San Mateo: Morgan Kaufmann, 113-118.

26. Muggleton, S., and Feng, C. (1992). Efficient induction of logic programs. In S. Muggleton (Ed.), *Inductive Logic Programming*, 281-298. London: Academic Press.

27. Muggleton, S, King, R.D., and Sternberg, M.J. (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering, 5*, 646-657.

28. Pazzani, M.J., Brunk, C.A., and Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. *Proceedings Eighth International Workshop on Machine Learning*, Evanston, Illinois, 432-436. San Mateo: Morgan Kaufmann.

29. Pazzani, M.J., and Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning, 9*, 1, 57-94.

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

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

32. Quinlan, J.R. (1991). Determinate literals in inductive logic programming. *Proceedings Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, 746-750. San Mateo: Morgan Kaufmann.

33. Quinlan, J.R. (1993). *C4.5: Programs for Machine Learning.* San Mateo: Morgan Kaufmann.

34. Quinlan, J.R., and Cameron-Jones, R.M. (1993). FOIL: a midterm report. *Proceedings European Conference on Machine Learning*, Vienna, 3-20. Berlin: Springer-Verlag.

35. 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.

36. Sammut, C.A. (1993). The origins of inductive logic programming: a prehistoric tale. *Proceedings Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, 127-147.

37. Semeraro, G., Brunk, C.A., and Pazzani, M.J. (1993). Traps and pitfalls when learning logical theories: a case study with FOIL and FOCL. Technical Report 93-33, Department of Information and Computer Science, University of California, Irvine, USA.

38. Shapiro, E.Y. (1983). *Algorithmic Program Debugging.* Cambridge, MA: MIT Press.

39. Winston, P.H. (1975). Learning structural descriptions from examples. In P.H. Winston (Ed), *The Psychology of Computer Vision.* New York: McGraw-Hill.

40. Wogulis, J., and Pazzani, M.J. (1993). A methodology for evaluating theory revision systems: results with AUDREY II. *Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1128-1134. San Francisco: Morgan Kaufmann.

41. Zelle, J.M., and Mooney, R.J. (1993). Combining FOIL and EBG to speed-up logic programs. *Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1106-1111. San Francisco: Morgan Kaufmann.

42. Zelle, J.M., and Mooney, R.J. (1994). Inducing deterministic Prolog parsers from Treebanks: a machine learning approach. *Proceedings AAAI-94 Twelfth National Conference on Artificial Intelligence*, Seattle, Washington. Menlo Park: AAAI Press.

43. Zelle, J.M., Mooney, R.J., and Konvisser, J.B. (1994). Combining top-down and bottom-up techniques in inductive logic programming. *Proceedings Eleventh International Conference on Machine Learning*, New Brunswick, New Jersey, 343-351. San Francisco: Morgan Kaufmann.