Determinate Literals in Inductive Logic Programming*

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

Abstract

A recent system, FOIL, constructs Horn clause programs from numerous examples. Computational efficiency is achieved by using greedy search guided by an information-based heuristic. Greedy search tends to be myopic but *determinate terms,* an adaptation of an idea introduced by another new system (GOLEM), has been found to provide many of the benefits of lookahead without substantial increases in computation. This paper sketches key ideas from FOIL and GOLEM and discusses the use of determinate literals in a greedy search context. The efficacy of this approach is illustrated on the task of learning the quicksort procedure and other small but non-trivial list-manipulation functions.

1 Introduction

The learning task of discovering Horn clause programs from examples has been studied for some time, with important contributions from Plotkin [1971], Shapiro [1983], Sanimut and Banerji [1986], Muggleton [1987], Buntine [1988], and Muggleton and Buntine [1988]. As Mitchell [1979] points out, all inductive learning requires search of a space of possible hypotheses. Even though Horn clause programs are a restricted class of first-order formalisms, the hypothesis space is so large that such learning programs often required carefully selected examples and/or hints of one kind or another in order to discover useful programs in a reasonable time.

Recent papers [Quinlan, 1990a, 1990b] introduce a new system, FOIL, that adapts ideas from attribute-value learning to this task. Specifically, FOIL exploits information from large numbers of examples to guide the search for a program. Such guidance turns out to be so effective that greedy search is usually adequate, permitting common benchmark problems to be solved very quickly. [Quinlan, 1990a] contains examples drawn from six task domains studied previously by Machine Learning researchers, all of which FOIL handles competently.

*The research reported here was supported by a grant from the Australian Research Council.

Algorithms using greedy search, however, tend to suffer from a horizon effect an action that may prove to be desirable or even essential from a global perspective may appear relatively unpromising at a local level and so may be passed over. As a result, even though the original FOIL (FOIL.0) performs well on common tasks, there are some programs that it will not discover.

An even more recent system, GOLEM [Muggleton and Feng, 1990], takes a very different approach to this same learning task. Using Plotkin's concept of generalisation lattices, GOLEM generates clauses as the least general generalisation of two examples in the context of available background knowledge. These generalised clauses can be unmanageably large and, to restrain their size, GOLEM considers only clauses in which all terms are *determinate* in the sense that there is only one binding possible for existentially quantified variables. With this restriction on what can be learned, GOLEM solves some very difficult tasks involving complex clauses and numerous examples and, like FOIL, does so very efficiently.

This paper concerns an adaptation of Muggleton's determinate terms, called *determinate literals.* Rather than restricting the search space, and thus the class of learnable programs, FOIL exploits the idea of determinism to overcome some of the horizon effect of greedy search. The effect on learning time is usually negligible - in fact, some learning problems are now solved more quickly than before.

After an abbreviated description of some key ideas underlying FOIL and GOLEM, the paper describes determinate literals and their use in a greedy search context. The usefulness of these determinate literals is illustrated on the task of learning the quicksort procedure and other small list-manipulation programs.

2 FOIL

FOIL's input consists of information about one or more relations, one of which (the *target relation)* is to be defined by a Horn clause program. For each relation we are given a set of tuples of constants that are in the relation. For the target relation we might also be given tuples that are known not to belong to the relation; alternatively, the closed world assumption may be invoked to state that no tuples, other than those specified, belong to the target relation. Tuples belonging to the target relation are labeled ⊕, those not belonging to it ⊖. The learning task is then to find a set of clauses for the target relation that account for all the ⊕ tuples while not covering any of the ⊖ tuples.

This description is somewhat over-simplified. In many real-world situations, noise in the data prevents learning exact, complete definitions of this form. To get around this problem, FOIL uses encoding-length heuristics to limit the complexity of clauses and programs; the underlying idea is that the number of bits required to represent a clause should be less than the number of bits required to represent the tuples it covers. The final clauses may cover most (rather than all) of the ⊕ tuples while covering few (rather than none) of the ⊖ tuples. See [Quinlan, 1990a] for details.

The basic approach used by FOIL is an AQ-like covering algorithm [Michalski, 1980]. We start with a *training set* containing all ⊕ and ⊖ tuples, construct a function-free Horn clause to 'explaîn^7 some of the 0 tuples, remove the covered 0 tuples from the training set, and continue with the search for the next clause.

This paper focuses on the construction of a single clause. FOIL starts with the left-hand side and extends the clause by adding literals to the right-hand side, stopping when no ⊖ tuples are covered by the clause (or when encoding-length heuristics decide that the clause is too complex). Although FOIL incorporates a simple backup mechanism, the clause-building process is essentially a greedy search; once a literal is added to a clause, alternative literals are usually not investigated. We look now at how this literal is chosen at each step.

Consider the partially developed clause

$$A \leftarrow L_1, L_2, \ldots, L_{m-1}$$

containing variables $V_1, V_2, \ldots, V_x$. Each tuple in the training set T looks like $\langle c_1, c_2, \ldots, c_x \rangle$ for some constants {c,}, and represents a ground instance of the variables in *A.* Now, consider what happens when a literal L_m of the form

$$P(V_{i_1}, V_{i_2}, \ldots, V_{i_p})$$

is added to the right-hand side of *A* giving a new clause *A'.* If the literal contains one or more new variables, the arity of the new training set will increase; let *x'* denote the number of variables in *A'.* Then, each tuple in the new training set T^1 will be of the form *(d_1, d_2 •••, $d_{x'}$)* for constants {cf,}, and will have the following properties:

- *{d_{i1}, d_{2l} ..., d_x)* is a tuple in T, and
- *($di_{1 1}d_{i2}$* ...,d_{ip}) is in the relation *P.*

That is, each tuple in *T'* is an extension of one of the tuples in T, and the ground instance that it represents satisfies the literal- Every tuple in *T* thus gives rise to zero or more tuples in T", the ⊕ or ⊖ labels of the new tuples being the same as their respective ancestors.

Let T_+ denote the number of O tuples in T, and similarly for T'_+. The effect of adding a literal L_m can be assessed from an information perspective as follows. The information conveyed by the knowledge that a tuple in *T* is ⊕ is given by

$$I(T) = -log_2(T_+ \ / \ |T|)$$

and similarly for $I(T')$. If $I(T')$ is less than *I(T)* we have 'gained^1 information by adding the lit .eral L_{in} to the clause and, if s of the tuples in *T* have extensions in T', the total information gained about the O tuples in *T* is

$$gain(L_m) = s \times (I(T) - I(T')).$$

FOIL explores the space of possible literals that might be added to a clause at each step, looking for the one with greatest positive gain. The form of the gain allows significant pruning of the literal space, so that FOIL can usually rule out large subspaces without having to examine any literals in them.

FOIL thus tames the hypothesis space problem by a stepwise greedy search for clauses. However, some clauses in reasonable definitions will inevitably contain literals with zero (or negative) gain. Suppose, for instance, that all objects have a value for some property *Q,* and the literal *Q(X,Y)* defines the value *Y* for object A. Since this literal represents a one-to-one mapping from *X* to *Y*, each tuple in *T* will give rise to exactly one tuple in *T'* and so the literal's gain will always be zero. To permit such literals to appear in clauses, FOIL ascribes a small positive gain to any literal that introduces a new variable. This has the effect of widening the search space for the next literal - there will be more possible combinations of arguments for the literal - but also can cause a blow-out in the size of the training set of tuples. Worse, since there are usually many literals that would introduce new variables, the choice of one of them is necessarily arbitrary.

3 GOLEM and Determinate Terms

GOLEM uses the same information as FOIL, namely constant tuples that belong to one or more relations, but regards each tuple as a ground assertion. If a tuple $\langle c_1, c_2, \ldots, c_n \rangle$ is a member of relation *R,* this is equivalent to the ground assertion $R(c_1, c_2, \ldots, c_n)$.

Let e_1 and e_2 be two terms. Plotkin's least general generalisation of e_1 and e_2, denoted *lgg(e_1,e_2),* is defined as follows [Muggleton and Feng, 1990]:

- if $e_1 = e_2$, $lgg(e_1, e_2) = e_1$
- if e_1 is $f(s_1, s_2, \ldots, s_n)$ and e_2 is $f(t_1, t_2, \ldots, t_n)$, $lgg(e_1, e_2) = f(lgg(s_1, t_1), lgg(s_2, t_2), \ldots, lgg(s_n, t_n))$
- otherwise, $lgg(e_1, e_2)$ is a variable V, where the same variable is used everywhere as the lgg of these terms.

Now, suppose we have two ground examples of the target relation, *R(c_1,* c_2, .., *c_n)* and *R(d\$_y$ $d_{2)}$..., d_n),* say, and a number of other relations representing background knowledge. The least general generalisation of these two examples relative to the background knowledge, the *?rl-ativc* least general generalisation (rlgg), is a clause

$$R(lgg(c_1, d_1), lgg(c_2, d_2), \ldots) \leftarrow \wedge\{lgg(x_1, x_2)\}$$

for every pair x_1, x_2 of ground assertions taken from each relation. The right-hand side of this clause is equivalent to *true* if the examples cannot be covered by a single clause, or, if they can, consists of a (usually very large) number of literals. The basic operation of GOLEM can be thought of as choosing a pair of examples in the target relation and forming their rlgg, successively expanding the right-hand side until the clause covers only positive examples of the target relation. If the resulting clause is not vacuous it is refined by removing unnecessary literals and added to the developing program in a manner similar to FOIL.

Consider now the clause

$$A \leftarrow L_1, L_2, \dots, L_m$$

and suppose t is a term in L_m. Denote by *{Y}* the set of variables that occur only in *t* and by *{Z}* the other variables in the clause. Term *t* is *determinate* wrt L_m if, for every ground substitution for variables in *{Z}*, there is at most one ground substitution for variables in *{Y)* so that the clause is satisfied. Intuitively, the values of variables in *{Y}* are determined by the values of variables that appear earlier in the clause.

GOLEM uses this idea to eliminate most literals that might be added to the right-hand side of a clause by insisting that, in a new literal, every term containing a new variable must be determinate wrt that literal. Although this rules out some potential clauses, the reduction in the number of literals in the rlgg of two examples is so substantial that rlgg's can be constructed feasibly and efficiently.

4 Determinate Literals

The key idea in determinate terms is that new variables have a value forced by previous variables. *Determinate literals* employ the same intuition, but also insist on comprehensive coverage of the O tuples in the current training set.

Suppose that we have an incomplete clause

$$A \leftarrow L_1, L_2, \dots, L_{m-1}$$

with an associated training set *T* as before. A literal 'L_m is determinate with respect to this partial clause if L_m contains new variables and there is exactly one extension of each O tuple in T, and no more than one extension of each 0 tuple, that satisfies L_m. The idea is that, if L_m is added to the clause, no $\oplus$ tuple will be eliminated and the new training set *T'* will be no bigger than T.

FOIL notes determinate literals found while exploring possible literals to add to the clause. The maximum possible gain is given by a literal that excludes all $\ominus$ tuples and no $\oplus$ tuples; in the notation used before, this gain is $T_+ \times I(T)$. Unless a literal is found whose gain is close to ($\geq$ 80% of) the maximum possible gain, FOIL adds *all* determinate literals to the clause and tries again. This may seem rather extravagant, since it is unlikely that all such literals will be useful. However, FOIL incorporates clause-refining mechanisms that remove unnecessary literals as each clause is completed, so there is no ultimate penalty for this all-in approach. Since no 0 tuples are eliminated and the training set does not grow, the only computational cost is associated with the introduction of new variables and the corresponding increase in the space of possible next literals. It is precisely the enlargement of this space that the addition of determinate literals is intended to achieve.

There is, of course, a potential runaway situation in which determinate literals found at one cycle give rise to further determinate literals at the next *ad infinitum.* To circumvent this problem, FOIL borrows another idea from GOLEM. The *depth* of a variable is determined by its first occurrence in the clause. All variables in the left-hand side of the clause have depth 0; a variable that first occurs in some literal has depth one greater than the greatest depth of any previously-occurring variable in that literal. By placing an upper limit on the depth of any variable introduced by a determinate literal, we ensure that there are a finite number of them and so rule out indefinite runaway. This limit does reduce the class of learnable programs. However, the stringent requirement that a determinate literal must be uniquely satisfied by *all* 0 tuples means that this runaway situation is very unlikely and FOIL's default depth limit (5) is hardly ever reached.

5 An Example: Quicksort

One difficult task investigated by Muggieton is learning the quicksort procedure for sorting lists of numbers. There are six relations:

sort(U,S)	sorting list U gives list S
partition(V,U,L,H)	partitioning list U on value V gives sublist L (values $\leq$ V) and sublist H (values > V)
components(L,H ,T)	list L has head H and tail T
append (A,B,C)	appending list A to list B gives list C
null(L)	L is the empty list
elt(V)	V is a number

Following Muggleton, the examples provided for the *sort* relation cover all lists of length up to three containing non-repeated numbers in {0,1,2}. There are 16 such lists, giving 16 O and 240 $\ominus$ tuples. The other relations are defined over this same vocabulary.

The definition found by FOIL in 10.3 seconds (on a DECstation 3100) is

$$\begin{aligned} &sort(A,B) \leftarrow\ =(A,B),\ null(A) \\ &sort(A,B) \leftarrow\ components(A,C,D), \\ &\qquad partition(C,D,E,F), \\ &\qquad sort(E,G),\ sort(F,H), \\ &\qquad append(G,I,B), \\ &\qquad components(I,C,H) \end{aligned}$$

The first four literals of the second clause have negligible or zero gain. However, all these (and the fifth literal) are determinate literals, as can be seen by observing that

Elements	Max Size	Lists	Tuples ($\oplus$ and $\ominus$)	Tuple Ratio	Time (secs)	Time Ratio
$\{0,1,2\}$	3	16	256	1.0	10.3	1.0
$\{0,1,2,3\}$	3	41	1681	6.6	69.4	6.7
$\{0,1,2,3\}$	4	65	4225	16.5	177.9	17.3
$\{0,1,2,3,4\}$	4	206	42,436	165.8	1305.1	126.7

Table 1: Results on Quicksort Experiments

- in *components(A, C,D), C* and *D* are determined by *A-*
- in *partition(C,D,E,F), E* and *F* are determined by *C* and *D;*
- in *sort(E,G), E* determines *G*
- in *sort(F,H), F* determines *H,*
- in *append(G,I,B), I* is determined by *G* and *B.*

Consequently, all these are included in the developing clause. (Two further determinate literals, *components(B,X,Y)* and *sort(Y,Z),* were also added as the search progressed, and were discarded as unnecessary when the clause was completed.)

The effect of the determinate literals above is to introduce new variables that possess a useful relationship to the variables *A* and *B* in the left-hand side of the clause. Only when these variables are in place can FOIL 'see' the importance of the last couple of literals. Without determinate literals, FOIL does not discover the second recursive clause, but instead develops a swatch of less general clauses covering special cases (e.g., for lists of length one, for lists that are already sorted, and so on).

To illustrate FOIL's computational economy, the experiment was repeated using larger numbers of examples of the *sort* relation. Three additional data sets were defined by increasing the allowable set of elements of a list and by increasing the maximum length of the lists. As Table 1 shows, there are more lists and much larger numbers of tuples in these training sets, with a corresponding increase in the size of the other relations. The same definition for *sort* was found in each case and the time required grew approximately linearly with the size of the training set. On this task, FOIL does not suffer from a combinatorial increase in computation time as the number of training examples grows; the same finding has been made in other domains.

6 Summary of Other Results

FOIL has been applied to more than twenty tasks from ten domains ranging from learning approximate rules for a small chess task to discovering an unknown dealer's rule in Eleusis; several are discussed in [Quinlan, 1990a].

Here, we give results on several domains akin to quicksort, viz. that require the system to learn exact, recursive definitions. Many of these datasets were provided by Stephen Muggleton. A brief description of each follows:

- *member,* find whether a list or element is a member of another list, given a few examples. Other relations were *components* and *null.*
- *append:* find how to append one list to another. Examples covered all lists of length up to three with non-repeating elements drawn from {a, 6, c}; a sample of about, 10K negative examples was provided- Other relations were *components, null, list* and *reverse* - note that the last two of these were not required for the definition of *append.*
- *reverse,* find how to reverse a list. Examples were the same as for the *append* relation above.
- *combinations:* find a recurrence relation for the number of combinations of *r* objects chosen from n, given all values with $r \leq n \leq 5$. Other relations were *multiply, zero* and *one.*
- *multiply:* find a recurrence relation for multiplication given the table of values up to *6x6.* Other relations were *predecessor, plus, zero* and *one.*

Results on these domains are presented in Table 2, first using determinate literals and then with this feature disabled. As before, times are for a DECstation 3100.

In two cases when determinate literals were not used, the definitions found by FOIL were imperfect in that they did not cover the whole training set, or covered it in a non-general way. In *reverse* and *multiply* the learning time was actually reduced by the use of determinate literals. In the case of *append,* on the other hand, there was a three-fold increase in time required. As this last example represents the downside of using determinate literals, let us have a closer look at it.

Recall that the relations defined for *append* include the (unrelated) relation *reverse.* When determinate literals are employed, the development of a clause for *append(A,B,C)* immediately gives three determinate literals:

reverse (A,D)
reverse(B,E)
reverse (C,F)

(since each argument has a unique reverse). Now, instead of three bound variables, there are six! If we continue by examining possible literals based on the *components* relation, with three arguments, there are 42 possible literals that need to be looked at. On the other hand, if the determinate (but unhelpful) literals are not added, there are only 12 *components* literals that must be assessed. A scale-up of the same kind occurs for other literals. Thus, while the search for clauses in the two

Domain	With Determinate Literals	Without Determinate Literals
sort	10.3	-
member	0.1	0.1
append	84.1	26.4
reverse	30.5	157.3
combinations	17.2	-
multiply	4.1	8.4

Table 2: Times for Other Domains ('-' means imperfect definition)

cases follows somewhat similar paths, the introduction of additional, red-herring variables by the determinate literals leads to more literal evaluations. The increased learning time is a direct result.

In many cases, the use of determinate literals produces no noticeable effect on the time required to find a set of clauses, and only occasionally increases the time required to find a solution. On the other hand, the introduction of new variables can materially increase FOIL's ability to find a solution when long, recursive clauses must be generated.

7 Conclusion

This paper has presented some highlights of two new systems for learning Horn clause programs from examples. One of them, FOIL, uses a greedy search to explore the very large hypothesis spaces involved, while the other, GOLEM, employs an interesting semantic concept to limit this space.

An adaptation of this concept, determinate literals, has been shown to overcome some of the horizon problems associated with greedy search. The application of this idea does not appreciably limit the class of programs that can be learned; instead, it has the effect of introducing a kind of low-cost lookahead in situations where there is not a very clear indication of what to do next.

GOLEM and FOIL are both young systems that are still evolving, so it is perhaps premature to compare them. However, informal trials have revealed a surprising degree of similarity in many of the solutions found by these systems and the computational effort required, despite the very different approaches that they embody.

Acknowledgements

I have benefited greatly from discussions with many colleagues, particularly Stephen Muggleton and Donald Michie. Stephen Muggleton provided some of the examples used in this paper. Some of this work was carried out while I was visiting the Knowledge Systems Laboratory, Stanford University.

References

Buntine, W. (1988). Generalised subsumption and its applications to induction and redundancy. *Artificial Intelligence 36,* 149-176.

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

Mitchell, T.M. (1979). An analysis of generalization as a search problem, in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence,* Tokyo, pp. 577-582. San Mateo: Morgan Kaufmann.

Muggleton, S.H. (1987). Duce, an oracle-based approach to constructive induction. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence,* Milan, pp. 287-292. San Mateo: Morgan Kaufmann.

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

Muggleton, S., and Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory,* Tokyo: Ohmsha.

Quinlan, JR. (1990a). Learning logical definitions from relations. *Machine Learning* 5, 239-266.

Quinlan, J.R. (1990b). Learning from relational data. In *Proceedings of the Fourth Australian Joint Conference on Artificial Intelligence,* Perth, pp. 38-47. Singapore: World Scientific.

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.

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