# An ILP System for Learning Head Output Connected Predicates

José C.A. Santos, Alireza Tamaddoni-Nezhad, and Stephen Muggleton

Department of Computing, Imperial College London
{jcs06,atn,shm}@doc.ic.ac.uk

**Abstract.** Inductive Logic Programming (ILP) [1] systems are general purpose learners that have had significant success on solving a number of relational problems, particularly from the biological domain [2,3,4,5]. However, the standard compression guided top-down search algorithm implemented in state of the art ILP systems like Progol [6] and Aleph [7] is not ideal for the Head Output Connected (HOC) class of learning problems. HOC is the broad class of predicates that have at least one output variable in the target concept. There are many relevant learning problems of this class such as arbitrary arithmetic functions and list manipulation predicates which are useful in areas such as automated software verification[8]. In this paper we present a special purpose ILP system to efficiently learn HOC predicates.

## 1 Introduction

Inductive Logic Programming (ILP) [1] systems are general purpose machine learners that have had significant success on solving a number of relational problems particularly from the biological domain [2,3,4,5].

However, the standard compression guided top-down search algorithm implemented in state of the art ILP systems like Progol [6] and Aleph [7] is not ideal for the Head Output Connected (HOC) class of learning problems. HOC is the broad class of predicates that have at least one output variable in the target concept.

The reason compression or coverage guided searchs are not suitable for this class of problems is because it really only makes sense to evaluate the clause when the head output variable has been instantiated in the body of the clause. In the HOC class of problems knowning the coverage of the intermediate clauses, besides computationally expensive, is useless and cannot guide the search as often all the intermediate clauses cover all the examples or the coverage bears no relation to the target concept.

Thus, unfortunately, recent significant performance improvements [9] to general purpose ILP systems that focus on more efficient algorithms and datastructures to process the hypotheses and their coverages, albeit very useful in general, are of little help for the HOC class of problems.

The HOC class is a special case of relational path finding which was first studied by Richards and Mooney [10]. More recently, Ong et al. [11] extended Richards and Mooney's work to make use of mode declarations and bottom clauses. Our work also uses the mode declarations to specify the call modes (input or output) for predicate variables and the bottom clause to anchor one end of the search space but proposes an efficient algorithm for the case when there is at least one head output variable in the target concept to be induced.

The HOC class of problems may seem too specific at first but several important problems belong to it. Perhaps the most evident one are arbitrary arithmetic functions (e.g. polynomials, factorial, fibonnaci, . . . ). Being able to efficiently induce arithmetic functions is important in software verification, for instance, to find loop invariants [8].

This paper is organized as follows. In section 2, the minimal background in ILP is provided in order for this paper to be self contained. In section 3, the algorithm to efficiently learn problems in the HOC class is presented. In section 4 an empirical evaluation is presented. Finally we conclude on section 6.

## 2 ILP Background

We assume the reader to be familiar with the basic concepts from logic programming [12] and inductive logic programming [13]. This section is intended as a reminder of some of concepts used in definitions in this paper.

Several ILP systems, including Progol [6] and Aleph [7], use mode declarations to constrain the search space. Mode declaration ($M$), definite mode language ($\mathcal{L}(M)$) and depth-bounded mode language ($\mathcal{L}_i(M)$) used in these systems are defined as follows.

**Definition 1 (Mode declaration $M$).** *A mode declaration has either the form modeh(n,atom) or modeb(n,atom) where $n$, the recall, is either an integer, $n > 1$, or '*' and atom is a ground atom. Terms in the atom are either normal or place-marker. A normal term is either a constant or a function symbol followed by a bracketed tuple of terms. A place-marker is either +type, -type or #type, where type is a constant. If $m$ is a mode declaration then $a(m)$ denotes the atom of $m$ with place-markers replaced by distinct variables. The sign of $m$ is positive if $m$ is a modeh and negative if $m$ is a modeb.*

**Definition 2 (Definite mode language $\mathcal{L}(M)$).** *Let $C$ be a definite clause with a defined total ordering over the literals and $M$ be a set of mode declarations. $C = h \leftarrow b_1, .., b_n$ is in the definite mode language $\mathcal{L}(M)$ if and only if 1) $h$ is the atom of a modeh declaration in $M$ with every place-marker +type and -type replaced by variables and every place-marker #type replaced by a ground term and 2) every atom $b_i$ in the body of $C$ is the atom of a modeb declaration in $M$ with every place-marker +type and -type replaced by variables and every place-marker #type replaced by a ground term and 3) every variable of +type in any atom $b_i$ is either of +type in $h$ or of -type in some atom $b_j$, $1 \leq j < i$.*

**Definition 3 (Depth of variables).** *Let $C$ be a definite clause and $v$ be a variable in $C$. Depth of $v$ is defined as follows:*

$$d(v) = \begin{cases} 0 & \textit{if } v \textit{ is in the head of } C \\ (max_{u \in U_v} d(u)) + 1 & \textit{otherwise} \end{cases}$$

*where $U_v$ are the variables in atoms in the body of $C$ containing $v$.*

**Definition 4 (Depth-bounded mode language $\mathcal{L}_i(M)$).** *Let $C$ be a definite clause with a defined total ordering over the literals and $M$ be a set of mode declarations. $C$ is in $\mathcal{L}_i(M)$ if and only if $C$ is in $\mathcal{L}(M)$ and all variables in $C$ have depth at most $i$ according to Definition 3.*

Progol and Aleph search a bounded sub-lattice for each example $e$ relative to background knowledge $B$ and mode declarations $M$. The sub-lattice has a most general element which is the empty clause, $\square$, and a least general element $\bot_i$ which is the most specific element in $\mathcal{L}_i(M)$ such that

$$B \wedge \bot_i \wedge \overline{e} \vdash_h \square$$

where $\vdash_h \square$ denotes derivation of the empty clause in at most $h$ resolutions. The following definition describes a bottom clause $\bot_i$ for a depth-bounded mode language $\mathcal{L}_i(M)$.

**Definition 5 (Most-specific clause or bottom clause).** *Let $h, i$ be natural numbers, $B$ be a set of Horn clauses, $e = a \leftarrow b_1, .., b_n$ be a definite clause, $M$ be a set of mode declarations containing exactly one modeh $m$ such that $a(m) \succeq a$ and $\hat{\bot}$ be the most-specific (potentially infinite) definite clause such that $B \wedge \hat{\bot} \wedge \overline{e} \vdash_h \square$. $\bot_i$ is the most-specific clause in $\mathcal{L}_i(M)$ such that $\bot_i \succeq \hat{\bot}$. $C$ is the most-specific clause in $\mathcal{L}$ if for all $C'$ in $\mathcal{L}$ we have $C' \succeq C$.*

## 3 Head Output Connected Learning

In this section we introduce some definitions needed to present the HOC learning algorithm.

**Definition 6 (IO consistent).** *A definite clause $h \leftarrow b_1, .., b_n$ is said to be IO consistent iff the input variables for each body atom $b_i$, are either a subset of the head input variables or are found in a previous body atom $b_j$, with $1 \leq j < i$.*

*Example 1.* Given the predicate signatures c(+int), a(+int,-int), b(+int, -int), $c(X) \leftarrow a(X, Y), b(Y, Z)$ is IO consistent but $c(X) \leftarrow a(X, Y), b(Z, Y)$ is not because $Z$ is an input variable for $b/2$ but is not connected to the head input variables.

An ILP system only generates IO consistent clauses as valid hypotheses. By construction the bottom clause ensures it.

**Definition 7 (Head Output Connected (HOC)).** *A definite clause $C$ is said to be Head Output Connected iff it is IO consistent and all its head output variables are instantiated in the body (i.e. there is a chain of literals connecting the head input variables to the head output variables).*

*Example 2.* Given the predicate signatures c(+int,-int), a(+int,-int), b(+int, -int), $c(X, Y) \leftarrow a(X, Z), b(Z, Y)$ is Head Output Connected as there is a chain of literals from variable $X$ to variable $Y$.

Note that a general purpose ILP system does not distinguish HOC clauses from non-HOC clauses.

**Definition 8 (Support clause at depth $D$).** *A clause $C'$ is a support clause (SC) for a clause $C$ at depth $D$ iff, $C'$ subsumes $C$, $C'$ has $D$ literals, is Head Output Connected and has itself no shorter support clause (i.e. $SC(C') = C'$).*

Support clauses are an important concept in the context of head output variable connectness. They are, at a given depth, the shortest clauses that minimally connect the clause's head input variables to its head output variables.

Note that the support clause at a given depth is not necessarily unique as Example 3 illustrates.

*Example 3.* Let $Clause = a(A, B) \leftarrow b(A, C), b(A, B), c(A, D), c(C, B), c(D, B)$ and the mode declarations are:

$\{a(+t, -t), b(+t, -t), c(+t, -t)\}$, with $t$ an arbitrary type shared by all predicates (e.g. int), then:
$SC_{Clause,1} = \{\{a(A, B) \leftarrow b(A, B)\}\}$
$SC_{Clause,2} = \{\{a(A, B) \leftarrow b(A, C), c(C, B)\}, \{a(A, B) \leftarrow b(A, D), c(D, B)\}\}$
$SC_{Clause,3} = \{\}$

Note that HOC learning is robust to both determinate and non-derminate predicates, both in head or in the body of a clause (b/2 and c/2 are non-determinate in the above example).

In order to search for the Head Output Connected clauses we need to define an ordering of clauses from the Head to the most specific clause, $\perp$. It is useful to first define the concept of subclause 9.

**Definition 9 (Subclause of another clause).** *Let $C = h \leftarrow b_1, .., b_n$ be an arbitrary IO consistent clause. $C'$ is a subclause of $C$ iff it is of the form $h \leftarrow b'_1, .., b'_{n'}$, where $b'_1, .., b'_{n'}$ is a subsequence of $b_1, .., b_n$ and $C'$ is also an IO consistent clause.*

*Example 4.* Let $C = c(X) \leftarrow a(X), b(X), d(X)$, $C' = c(X) \leftarrow a(X), b(X)$ and $C'' = c(X) \leftarrow b(X), a(X)$, then $C'$ is a subclause of $C$ but $C''$ is not because its literals are not a subsequence of $C$.

**Definition 10 (Successors of a subclause with respect to a parent clause).** *A clause $C_{suc}$ is the sucessor of a clause $C_{sub}$ with respect to a clause $C$, iff $C_{sub}$ is a subclause of $C$, $C_{sub}$ is a subclause of $C_{suc}$ and $C_{suc}$ is one of the subclauses of $C$ obtained by adding a single atom from the body of $C$ to $C_{sub}$. The set of successors of a subclause $C_{sub}$ with respect to a parent clause $C$ (i.e. $S_{C_{sub},C}$), are all the $C_{suc}$ clauses that can be generated from this definition.*

*Example 5.* Let $C = c(X, Y) \leftarrow a(X, A), b(A, B), d(A, D), e(D, Y)$ and $C_{sub} = c(X, Y) \leftarrow a(X, A)$, then $C_{suc_1} = c(X, Y) \leftarrow a(X, A), b(A, B)$ and $C_{suc_2} = c(X, Y) \leftarrow a(X, A), d(A, D)$. Thus $S_{C_{sub},C} = \{C_{suc_1}, C_{suc_2}\}$. Notice that $S_{C_{suc_1},C} = \{\}$ and $S_{C_{suc_2},C} = \{c(X, Y) \leftarrow a(X, A), d(A, D), e(D, Y)\}$.

### 3.1 Algorithms

The algorithm to compute all the sets of support clauses for the head output variables until depth $D$ is given in Figure 1. It is essentially a breadth-first search over the implicit hypotheses space defined by the bottom clause. This space is traversed using the successor subclauses of the current subclauses to visit. The initial clause to visit is the head. Notice that the main difference to Progol and Aleph is that no coverage computation needs to take place during the search.

```
Set of support clauses for a clause, $C$, until a given max depth, $D$
  Input: Clause $C$ and depth $D$
  Let ClausesToVisit = $C$'s head
  Let SuppClauses = {}
  Let $Depth = 1$
  while ClausesToVisit$\neq$ {} and $Depth < D$ do
    ClausesToVisitNext = {}
    for each clause $c \in$ ClausesToVisit do
      Let SuccClauses = Successors of $c$ with respect to $C$
      Let CurSuppClauses = Head Output connected clauses $\in$ SuccClauses
      SuppClauses = SuppClauses $\cup$ CurSuppClauses
      ClausesToVisitNext = ClausesToVisitNext $\cup$ SucessorClauses - CurSuppClauses
    end for
    ClausesToVisit = ClausesToVisitNext
    $Depth = Depth + 1$
  end while
  Output: SuppClauses
```

**Fig. 1.** Algorithm to generate the set of support clauses for a single clause at depth $D$

```
Head Output Connected special purpose ILP system
  Input: Positive examples $E^+$, mode declarations $M$, settings $S$
         and background knowledge $B$
  Let Hyps = {}
  for each example $e \in E^+$ do
    Let $C = \bot_e$
    Let $Hyp_e$ = support set of clauses for $C$ until depth=max. clause length
    Hyps = Hyps $\cup$ $Hyp_e$
  end for
  Let Theory = Greedy search on Hyps for the highest scoring clauses
  Output: Theory
```

**Fig. 2.** Main loop of the Head Output Connected special purpose ILP system

Figure 2 presents the main cycle of the Head Output Connected special purpose ILP system. Notice that this algorithm is robust to the order of the examples as it processes all the examples. Only at the end, after all HOC clauses from all the examples are known, it performs a greedy search to find the set of HOC clauses that maximizes a clause evaluation function.

By default the clause evaluation function is compression. The compression of a clause is given by: $Pos - Neg - Len$, where $Pos$ is the number of positive examples covered, $Neg$ is the number of negative examples covered and $Len$ the number of literals in the clause. The related coverage evaluation function is identical except that it does not take into account the length of the clause.

## 4 Empirical Evaluation

In this section we perform an empirical evaluation of the HOC learning algorithm. We implemented the algorithm in YAP 6 [14] inside GILPS, our ILP system developed from scratch publicly available at http://www.doc.ic.ac.uk/~jcs06, and compare it against both Progol and Aleph.

### 4.1 Materials

Aleph 5 [7], Progol 4.4 [6] and our implementation of the HOC learner were used to solve two well known math problems: Fibonnaci and Binomial Coefficient. Note that, obviously, these problems have both one output variable.

In the Fibonnaci problem we are trying to learn the $i^{th}$ element of the series. That is, we want the ILP system to induce a predicate that returns the correct Fibonnaci function value for an arbitrary index $i > 1$ in the series. For instance, $Fib_8 = 21$.

$$Fib_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ Fib_{n-1} + Fib_{n-2}, & n \geq 2 \end{cases} \tag{1}$$

In the Binomial Coefficient the aim is identical, although the problem looks a bit more difficult as there are two input variables. We want the ILP system to induce a predicate that, given distinct non zero $n$ and $k$, returns the correct Binomial Coefficient. For instance, $\binom{6}{2} = 15$.

$$\binom{n}{k} = \begin{cases} 1, & n = k \vee k = 0 \\ \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & otherwise \end{cases} \tag{2}$$

Note that these are notouriously difficult problems for an ILP system to learn as the layer of new variables (parameter $i$ in Aleph and Progol) must be set to 3 and 4 (see 4.3), thus yielding a very large search space.

Figures 3 and 4 show the background knowledge and mode definitions used in Aleph. For the other ILP systems the same background knowledge and mode definitions were used apart from minor syntactic changes.

For both problems we provided, to all ILP systems, the base, non recursive cases, as part of the background knowledge. Only positive examples were given

```
:- modeh(1, fib(+int,-int)).
:- modeb(1, pred(+int,-int)).
:- modeb(1, fib(+int,-int)).
:- modeb(1, plus(+int,+int,-int)).
:- commutative(plus/3).

:- determination(fib/2, pred/2).
:- determination(fib/2, fib/2).
:- determination(fib/2, plus/3).

pred(A, B):- B is A-1.
plus(A, B, C):- C is A+B.

fib(0, 0):-!.
fib(1, 1):-!.
```

**Fig. 3.** Fibonnaci problem definition in Aleph

```
:- modeh(1, binomial(+int,+int,-int)).
:- modeb(1, pred(+int,-int)).
:- modeb(1, binomial(+int,+int,-int)).
:- modeb(1, plus(+int,+int,-int)).
:- commutative(plus/3).

:- determination(binomial/3, pred/2).
:- determination(binomial/3, binomial/3).
:- determination(binomial/3, plus/3).

pred(A, B):- B is A-1.
plus(A, B, C):- C is A+B.

binomial(_, 0, 1):-!.
binomial(N, N, 1):-!.
```

**Fig. 4.** Binomial Coefficient problem definition in Aleph

in both cases. When there are output variables in the head of the target concept, negative examples play a less important role as maximizing recall (i.e. just covering the positives) may be enough to find the correct definition.

The positive examples given were, for Fibonnaci: fib(7,13), fib(8,21), fib(9,34) and for Binomial: binomial(6,2,15), binomial(6,3,20) and binomial(7,3,35).

## 4.2 Methods

The three systems were run with identical settings for all parameters: number of nodes (5,000), depth of new variables (i=4 for Fibonnaci and i=3 for binomial), noise (0%) and maximum clause length (6).

Since we provide few examples the evaluation function was set to coverage in both Aleph and the HOC learner. In Progol, since it is not possible to have other evaluation function other than compression, the examples were inflated 5 times to achieve an effect identical to the usage of coverage as the evaluation function.

Progol was asked to generate a theory using the 'generalise' command and Aleph with the 'induce' command.

### 4.3 Results

Only the HOC ILP system was able to successfully learn the target concept. It took 0.4s to learn Fibonnaci and 2.1s to learn Binomial. Progol, even when nodes was later set to 100,000 could not learn either and took several hours to report it. Aleph crashed in both problems, after a few seconds[1].

```
fib(A,B) :-                            binomial(A,B,C) :-
   pred(A,C),    % i=1                    pred(A,D),          % i=1
   pred(C,D),    % i=2                    pred(B,E),          % i=1
   fib(C,E),     % i=3                    binomial(D,B,F),    % i=2
   fib(D,F),     % i=3                    binomial(D,E,G),    % i=2
   plus(F,E,B).  % i=4                    plus(G,F,C).        % i=3
```

**Fig. 5.** Induced Fibonnaci and Binomial Coefficient predicates

Figure 5 shows the induced predicates by the HOC ILP system. They are the intended definitions. We also added the $i$ setting, the layers of new variables introduced, which is a direct measure of the concept depth and an indirect measure of its complexity.

Notice that by default $i = 2$ in both Aleph and Progol, which is enough for typical classification problems [2,3,4,5] but insufficient to many HOC problems.

## 5 Discussion and Future Work

The HOC learning algorithm only solves one of the problems general purpose ILP systems have with this class of problems: it does not have to compute the coverage of the intermediate, non head output connected, clauses. However this approach still shares two problems: 1) computing the bottom clause which, depending on the mode declarations and background knowledge, can be quite large and 2) do a breadth-first search to find the support clauses for the head output variables.

[1] It was an "Instantiation error" while evaluating recursive clauses like "fib(A,B) :- pred(A,C), fib(C,D), pred(D,E)." We suppose this is a bug in Aleph while learning recursive clauses where the last literal is not the recursive call. We believe that even without this error Aleph would not be able to learn the target concepts as it uses essentially the same algorithm as Progol.

The breadth-first search could be relaxed and replaced by an heuristic that returns support clauses. However, in doing so, we could not guarantee that the shortest set of support clauses for a given clause length would be returned. Furthermore, while the breadth-first search is robust to any number of head output variables in the target concept, such an heuristic would not be.

As for the potentially very large bottom clauses, a problem shared with other mode directed inverse entailment ILP systems, it might be possible to avoid its generation at the start of the search if we perform a depth-first search and dynamically compute the sucessors of a clause, trading space by time.

Finally, it is worth pointing out that a simple inspection on the mode declarations is enough to automatically decide if the HOC learning algorithm should be used instead of the general purpose one. Therefore this algorithm can be easily integrated in general purpose ILP learners.

## 6 Conclusions

In this paper we presented a special purpose ILP algorithm to efficiently solve the class of problems where the concept to be learned has at least one output variable. This is an important class of problems that deserves special attention. We presented and implemented an algorithm that efficiently learns problems from this class. We showed two examples, Fibonnaci and Binomial coefficient, that can be learned in a coupple of seconds by our system but cannot be easily learned by a general purpose ILP system. The proposed approach is an initial step in handling more efficiently this class of problems in ILP systems.

## Acknowledgments

The first author thanks Wellcome Trust by the financial support for his Ph.D. scholarship. The second author was supported by the BBSRC CISBIC grant BB/C519670/1. The third author thanks the Royal Academy of Engineering and Microsoft for funding his present 5 year Research Chair. We are also indebted to three anonymous referees for valuable comments.

## References

1. Muggleton, S.: Inductive logic programming. New Generation Computing 8(4), 295–318 (1991)
2. Finn, P., Muggleton, S., Page, D., Srinivasan, A.: Pharmacophore discovery using the inductive logic programming system progol. Machine Learning 30, 241–271 (1998)
3. Srinivasan, A., King, R.D., Muggleton, S.H., Sternberg, M.: Carcinogenesis predictions using ilp. In: Džeroski, S., Lavrač, N. (eds.) ILP 1997. LNCS(LNAI), vol. 1297, pp. 273–287. Springer, Heidelberg (1997)
4. Srinivasan, A., Muggleton, S., King, R., Sternberg, M.: Theories for mutagenicity: a study of first-order and feature based induction. Artificial Intelligence 85(1,2), 277–299 (1996)

5. Turcotte, M., Muggleton, S.H., Sternberg, M.J.E.: Protein fold recognition. In: Page, D.L. (ed.) ILP 1998. LNCS, vol. 1446, pp. 53–64. Springer, Heidelberg (1998)
6. Muggleton, S.: Inverse entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming 13(3-4), 245–286 (1995)
7. Srinivasan, A.: The Aleph Manual. University of Oxford (2007)
8. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)
9. Fonseca, N.A., Costa, V.S., Rocha, R., Camacho, R., Silva, F.: Improving the efficiency of inductive logic programming systems. Softw., Pract. Exper. 39(2), 189–219 (2009)
10. Richards, B.L., Mooney, R.J.: Learning relations by pathfinding. In: AAAI, pp. 50–55 (1992)
11. Ong, I.M., de Castro Dutra, I., Page, D.L., Santos Costa, V.: Mode directed path finding. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 673–681. Springer, Heidelberg (2005)
12. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
13. Nienhuys-Cheng, S.-H., de Wolf, R.: Foundations of Inductive Logic Programming. LNCS(LNAI), vol. 1228. Springer, Heidelberg (1997)
14. Santos Costa, V.: The life of a logic programming system. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 1–6. Springer, Heidelberg (2008)