Refining Complete Hypotheses in ILP

Ivan Bratko

University of Ljubljana, Faculty of Computer and Information Sc.
Trzaska 25, 1000 Ljubljana, Slovenia
bratko@fri.uni-lj.si

Abstract. Most ILP systems employ the covering algorithm whereby hypotheses are constructed iteratively clause by clause. Typically the covering algorithm is greedy in the sense that each iteration adds the best clause according to some local evaluation criterion. Some typical problems of the covering algorithm are: unnecessarily long hypotheses, difficulties in handling recursion, difficulties in learning multiple predicates. This paper investigates a non-covering approach to ILP, implemented as a Prolog program called HYPER, whose goals were: use intensional background knowledge, handle recursion well, and enable multi-predicate learning. Experimental results in this paper may appear surprising in the view of the very high combinatorial complexity of the search space associated with the non-covering approach.

1 Introduction

Most ILP systems employ the covering algorithm whereby hypotheses are induced iteratively clause by clause. Examples of such systems are Quinlan's FOIL [5], Grobelnik's Markus [2], Muggleton's PROGOL [3] and Pompe's CLIP [4]. The covering algorithm builds hypotheses gradually, starting with the empty hypothesis and adding new clauses one by one. Positive examples covered by each new clause are removed, until the remaining positive examples are reduced to the empty set, that is, the clauses in the hypothesis cover all the positive examples.

Typically the covering algorithm is greedy in the sense that on each iteration it chooses to add the clause that optimises some evaluation criterion. Such a clause tends to be optimal locally, with respect to the current set of clauses in the hypothesis. However there is no guarantee that the covering process will result in a globally optimal hypothesis. A good hypothesis is not necessarily assembled from locally optimal clauses. On the other hand, locally inferior clauses may cooperate well as a whole, giving rise to an overall good hypothesis.

Some typical problems due to the greedy nature of the covering algorithm are:

- Unnecessarily long hypotheses with too may clauses
- Difficulties in handling recursion
- Difficulties in learning multiple predicates simultaneously

S. Dzeroski and P. Flach (Eds.): ILP-99, LNAI 1634, pp. 44-55, 1999.

In view of these difficulties a non-covering approach where a hypothesis (with all its clauses) would be constructed as a whole, would seem to be a better idea. Of course a strong practical reason against this, and in favour of the covering approach, is the combinatorial complexity involved. The local optimisation of individual clauses is complex enough, so the global optimisation of whole hypotheses would seem to be out of question. Experimental results in this paper are possibly surprising in this respect.

In this paper we investigate a non-covering approach and study its performance on typical small ILP learning problems that require recursion. We develop such a non-covering algorithm, implemented as a Prolog program called HYPER (Hypothesis Refiner, as opposed to the majority of ILP programs that are "clause refiners"). The design goals of the HYPER program were:

- simple, transparent and short
- handle intensional background knowledge
- handle recursion well
- enable multipredicate learning
- handle reasonably well typical basic ILP benchmark problems (member/2, append/3, path/3, sort/2, arches/3, etc.) without having to resort to special tricks, e.g. unnatural declaration of argument modes

HYPER does not address the problem of noisy data. So it aims at inducing short hypotheses that are consistent with the examples, that is: cover all the positive examples and no negative one.

2 Mechanisms of HYPER

According to the design goal of simplicity, the program was developed in the following fashion. First, an initial version was written which is just a simple generator of possible hypotheses for the given learning problem (i.e. bakground predicates and examples). The search strategy in this initial version was simple iterative deepening. Due to its complexity, this straightforward approach, as expected, fails even for simplest learning problems like member/2 or append/3. Then additional mechanisms were added to the program, such as better search and mode declarations, to improve its efficiency. Thirteen versions were written in this way with increasingly better performance. Once a reasonable performance on the basic benchmark problems was obtained, the added mechanisms were selectively switched off to see which of them were essential. Eventually, the final version was obtained with the smallest set of added mechanisms which still performs reasonably well. The mechanisms that were experimentally found to be essential are described in detail below. Later we also discuss mechanisms that somewhat surprizingly proved not to be particularly useful.

2.1 Search

HYPER constructs hypotheses in the top-down fashion by searching a refinement tree in which the nodes correspond to hypotheses. A hypothesis H_0 in this tree has successors H_i, where hypotheses H_i are the least specific (in some sense) refinements of of H_0. Refinements are defined by HYPER's refinement operator described in the next section. Each newly generated hypothesis is more specific than or equal to its predecessor in the sense of theta-subsumption. So a hypothesis can only cover a subset of the examples covered by the hypothesis' predecessor. The learning examples are always assumed noise-free, and the goal of search is to find a hypothesis that is consistent with the examples. That is, it covers all the positive examples and no negative example. If a hypothesis is generated that does not cover all the positive examples, it is immediately discarded because it can never be refined into a consistent hypothesis. Excluding such hypotheses from the search tree reduces the tree considerably. During search, new hypotheses are not checked whether they are duplicates or in any sense equivalent to already generated hypotheses.

Search starts with a set of initial hypotheses. This set is the set of all possible bags of user-defined start clauses of up to the user-defined maximal number of clauses in a hypothesis. Multiple copies of a start clause typically appear in a start hypothesis. A typical start clause is something rather general and neutral, such as: append(L1, L2, L3).

HYPER performs a best-first search using an evaluation function that takes into account the size of a hypothesis and its accuracy in a simple way by defining the cost of a hypothesis H as:

$$\text{Cost}(H) = w_1 * \text{Size}(H) + w_2 * \text{NegCover}(H)$$

where NegCover(H) is the number of negative examples covered by H. The definition of 'H covers example E' in HYPER roughly corresponds to 'E can be logically derived from H'. There are however some essential procedural details described later. w_1 and w_2 are weights. The size of a hypothesis is defined simply as a weighted sum of the number of literals and number of variables in the hypothesis:

$$\text{Size}(H) = k_1 * \#\text{literals}(H) + k_2 * \#\text{variables}(H)$$

All the experiments with HYPER described later were done with the following settings of the weights: $w_1=1$, $w_2=10$, $k_1=10$, $k_2=1$, which corresponds to:

$$\text{Cost}(H) = \#\text{variables}(H) + 10 * \#\text{literals}(H) + 10 * \text{NegCover}(H)$$

These settings are ad hoc, but their relative magnitudes are intuitively justified as follows. Variables in a hypothesis increase its complexity, so they should be taken into account. However, the literals increase the complexity more, hence they contribute to the cost with a greater weight. A covered negative example contributes to a hypothesis' cost as much as a literal. This corresponds to the intuition that an

extra literal should at least prevent one negative example from being covered. It should be noted that these weights can be varied considerably withought affecting the search performance. For example, changing k_1 from 1 to 5 had no effect on search in the experiments.

2.2 Hypothesis Refinement

To refine a clause, perform one of the following:

1. Unify two variables in the clause, e.g. X1 = X2.
2. Refine a variable in the clause into a background term, e.g. replace variable L0 with term [X|L].
3. Add a background literal to the clause.

Some pragmatic details of these operations are as follows:

(a) The arguments of literals are typed. Only variables of the same type can be unified. The user defines background knowledge, including „back-literals“ and „back-terms“. Only these can be used in refining a variable (of the approapriate type) into a term, and in adding a literal to a clause.
(b) Arguments in back-literals can be defined as input or output. When a new literal is added to a clause, all of its input arguments have to be unified (non-deterministically) with the existing non-output variables in the clause (that is those variables that are assumed to have their values instantiated at this point of executing the clause).

To refine a hypothesis H_0, choose one of the clauses C_0 in H_0, refine clause C_0 into C, and obtain a new hypothesis H by replacing C_0 in H_0 with C. This says that the refinements of a hypothesis are obtained by refining *any* of its clauses. There is a useful heuristic that often saves complexity. Namely, if a clause is found in H_0 that alone covers a negative example, then only refinements arising from this clause are generated. The reason is that such a clause necessarily has to be refined before a consistent hypothesis is obtained. This will be referred to as "covers-alone heuristic".

This refinement operator aims at producing least specific specialisations (LSS). However, it really only approximates LSS. This refinement operator does LSS under the constraint that the number of clauses in a hypothesis after refinement stays the same. Without this restriction, an LSS operator should be more appropriately defined as:

$$\text{refs\_hyp}(H_0) = \{ H_0 - \{C_0\} \cup \text{refs\_clause}(C_0) \mid C_0 \in H_0 \}$$

where refs_hyp(H_0) is the set of all LSS of hypothesis H_0, and refs_clause(C_0) is the set of all LSS of clause C_0. This unrestricted definition of LSS was not implemented in HYPER for the obvious reason of complexity.

2.3 Interpreter for Hypotheses

To prove an example, HYPER uses intensional background knowledge together with the current hypothesis. To this end HYPER includes a Prolog meta-interpreter which does approximately the same as the Prolog interpreter, but takes care of the possibility of falling into infinite loops. Therefore the length of proofs is limited to a specified maximal number of proof steps (resolution steps). This was set to 6 in all the experiments mentioned in this paper. It is important to appropriately handle the cases where this bound is exceeded. It would be a mistake to interpret such cases simply as 'fail'. Instead, the following interpretation was designed and proved to be essential for the effectiveness of HYPER. The interpreter is implemented as the predicate:

prove(Goal, Hypo, Answer)

Goal is the goal to be proved using the current hypothesis Hypo and background knowledge. The predicate prove/3 always succeeds and Answer can be one of the following three cases:

Answer = yes if Goal is derivable from Hypo in no more than D steps (max. proof length)
Answer = no if Goal is not derivable even with unlimited proof length
Answer = maybe if proof search was terminated after D steps

The interpretation of these answers, relative to the standard Prolog interpreter, is as follows. 'yes' means that Goal under the standard interpreter would definitely succeed. 'no' means that Goal under the standard interpreter would definitely fail. 'maybe' means any one of the following three possibilities:

1. The standard Prolog interpreter (no limit on proof length) would get into infinite loop.
2. The standard Prolog interpreter would eventually find a proof of length greater than D.
3. The standard Prolog interpreter would find, at some length greater than D, that this derivation alternative fails. Therefore it would backtrack to another alternative and there possibly find a proof (of length possibly no greater than D), or fail, or get into an infinite loop.

The question now is how to react to answer 'maybe' when processing the learning examples. HYPER reacts as follows:

- When testing whether a positive example is covered, 'maybe' is interpreted as 'not covered'.
- When testing whether a negative example is not covered, 'maybe' is interpreted as not 'not covered', i.e. as 'covered'.

A clarification is in order regarding what counts as a step in a proof. Only resolution steps involving clauses in the current hypothesis count. If backtracking occurs before proof length is exceeded, the backtracked steps are discounted. When proving the conjunction of two goals, the sum of their proof lengths must not exceed max. proof length. Calls of background predicates, defined in Prolog, are delegated to the standard Prolog interpreter and do not incur any increase in proof length. It is up to the user to ensure that the standard Prolog interpreter does not get into an infinite loop when processing such "background" goals.

2.4 Example

HYPER is implemented in Prolog. In this implementation, a hypothesis is represented as a list of clauses. A clause is a list of literals accompanied by a list of variables and their types. For example:

```
[member( X0, [X1 | L1]), member( X0, L2)] /
[X0:item, X1:item, L1:list, L2:list]
```

corresponds to the Prolog clause:

```
member( X0, [X1 | L1])  :- member( X0, L2).
```

where the variables X0 and X1 are of type item, and L1 and L2 are of type list. The types are user-defined.

Figure 1 shows the specification accepted by HYPER of the problem of learning two predicates simultaneously: even(L) and odd(L), where even(L) is true if L is a list with an even number of elements, and odd(L) is true if L is a list with an odd number of elements. In this specification,

```
backliteral( even( L), [L:list], []).
```

means: even(L) can be used as a background literal when refining a clause. The argument L is of type list. L is an input argument; there are no oputput arguments. prolog_predicate(fail) means that there are no background predicates defined in Prolog. The predicate term/3 specifies how variables of given types (in this case 'list') can be refined into terms, comprising variables of specified types. Start hypotheses are all possible bags of up to max_clauses = 4 start clauses of the two forms given in Fig. 1. For this learning problem HYPER finds the following hypothesis consistent with the data:

```
even( [ ] ).
even([ A, B | C ] )   :-   even( C).
odd( [ A | B ] )   :-   even( B).
```

Before this hypothesis is found, HYPER has generated altogether 66 hypotheses, refined 16 of them, kept 29 as further candidates for refinement, and discarded the remaining 21 as incomplete (i.e. not covering all the positive examples). To reach the final hypothesis above from the corresponding start hypothesis, 6 refinement steps are required. The size of the complete refinement forest of up to 6 refinement steps from the start hypotheses in this case, respecting the type constraints as specified in Fig. 1, is 22565. The actual number of hypotheses generated during search was thus less than 0.3% of the total refinement forest to depth 6.

This learning problem can also be defined more restrictively by only allowing term refinements on lists to depth 1 only, thus suppressing terms like [X1,X2 | L]. This can be done by using type "list(Depth)" in the definition of refine_term/3 and start_clause/1 as follows:

```
term( list( D), [ X | L ], [ X:item, L:list(1) ] ) :-
   var( D).      % list(1) cannot be refined further!
term( list( D), [ ], [ ]).
start_clause( [ odd( L) ] / [ L:list( D) ] ).
start_clause( [ even( L) ] / [ L:list( D) ] ).
```

Using this problem definition, HYPER finds the mutually recursive definition of even/2 and odd/2:

```
even( [ ] ).
odd( [A | B ] )   :-   even( B ).
even( [ A | B ] )   :-   odd( B ).
```

2.5 Mechanisms that Did Not Help

Several mechanisms were parts of intermediate versions of HYPER, but were eventually left out because they were found not to be clearly useful. They did not significantly improve search complexity and at the same time incurred some complexity overheads of their own. Some of these mechanisms are mentioned below as possibly useful „negative lessons“:

- Immediately discard clause refinements that render the corresponding hypothesis unsatisfiable (i.e. cannot succeed even on the most general query).
- Checking for redundant or duplicate hypotheses where „duplicate“ may mean either literally the same under the renaming of the variables, or some kind of equivalence between sets of clauses and literals in the clauses, or redundancy based on theta-subsumption between hypotheses. One such idea is to discard those newly generated and complete hypotheses that subsume any other existing candidate hypothesis. This requires special care because the subsuming hypothesis may later be refined into a hypothesis that cannot be reached from other hypotheses. Perhaps surprizingly no version of such redundancy or duplicate test

was found that was clearly useful. Also it is hard to find a useful subsumption-based test that would correspond well to the procedurally oriented interpretation of hypotheses.

```
% Inducing odd and even length property for lists

% Background literals

backliteral( even( L), [ L:list], [ ]).
backliteral( odd( L), [ L:list], [ ]).

% Term refinements

term( list, [ X | L ], [ X:item, L:list]).
term( list, [ ], [ ]).

% Background predicates defined in Prolog

prolog_predicate( none).     % No background predicate in Prolog

% Start clauses

start_clause( [ odd( L) ] / [ L:list]).
start_clause( [ even( L) ] / [ L:list]).

% Positive examples

ex( even( [ ] ) ).
ex( even( [a,b] ) ).
ex( odd( [a] ) ).
ex( odd( [b,c,d] ) ).
ex( odd( [a,b,c,d,e] ) ).
ex( even( [a,b,c,d] ) ).

% Negative examples

nex( even( [a] ) ).
nex( even( [a,b,c] ) ).
nex( odd( [ ] ) ).
nex( odd( [a,b] ) ).
nex( odd( [a,b,c,d] ) ).
```

Fig. 1: Definition of the problem of learning even/1 and odd/1 simultaneously.

- „Closing“ induced clauses as in Markus [2] to avoid the problem with uninstanted outputs of a target predicate. Closing a hypothesis means to „connect“ all the output arguments to instantiated variables (i.e. unifying all the output arguments with other arguments). There are typically many ways of closing a hypothesis. So in evaluating a hypothesis, a „best“ closing would be sought (one that retains the completeness and covers the least negative examples). This is, however, not only combinatorially inefficient, but requires considerable elaboration because often the closing is not possible without making the hypothesis incomplete (closing is sometimes too coarse a specialisation step).

3 Experiments

Here we describe experiments with HYPER on a set of typical ILP test problems. All of them, except arches/3, concern relations on lists and require recursive definitions. The problems are:

`member(X,List)`
`append(List1,List2,List3)`
`even(List) + odd(List)` (learning simultaneously even and odd length list)
`path(StartNode,GoalNode,Path)`
`insort(List,SortedList)` (insertion sort)
`arch(Block1,Block2,Block3)` (Winston’s arches with objects taxonomy)
`invariant(A,B,Q,R)` (Bratko and Grobelnik's program loop invariant [1])

For all of these problems, correct definitions were induced from no more than 6 positive and 9 negative examples in execution times typically in the order of a second or a few seconds with Sicstus Prolog on a 160 MHz PC. No special attention was paid to constructing particularly friendly example sets. Smaller example sets would possibly suffice for inducing correct definitions. Particular system settings or mode declarations to help in particular problems were avoided throughout. Details of these experiments are given in Table 1. The definitions induced were as expected, with the exception of a small surprise for path/3. The expected definition was:

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

The induced definition was slightly different:

```
path( A, A, [A]).
path( C, D, [C, B | A]):-
   link( C, B), path( B, D, [E | A]).
```

This might appear incorrect because the last literal would normally be programmed as path(B, D, [B | A]), stating that the path starts with node B and not with the undefined E. However, the heads of both induced clauses take care of eventually instantiating E to B.

The main point of interest in Table 1 are the search statistics. The last five columns give: the refinement depth RefDepth (the number of refinement steps needed to construct the final hypothesis from a start hypothesis), the number of all generated hypotheses, the number of refined hypotheses and the number of candidate hypotheses waiting to be refined, and the total size of the refinement forest up to depth RefDepth. This size corresponds to the number of hypotheses that would have to be generated if the search was conducted in the breadth-first fashion. The number of all generated hypotheses is greater than the sum of refined and to-be-refined hypotheses. The reason is that those generated hypotheses that are not complete (do not cover all the positive examples) are immediately discarded. Note that all these counts include duplicate hypotheses because when searching the refinement forest the newly generated hypotheses are not checked for duplicates. The total size of the refinement forest is determined by taking into account the covers-alone heuristic. The sizes would have been considerably higher without this heuristic, as illustrated in Table 2. This table tabulates the size of the search forest and the size of the refinement forest by refinement depth, and compares these sizes with or without covers-alone heuristic and duplicate checking. The total sizes of the refinement forests in Table 1 were determined by generating these trees, except for append/3 and path/3. These trees were too large to be generated, so their sizes in Table 1 are estimates, obtained by extrapolating the exponential growth to the required depth. These estimates are considerable underestimates.

Tables 1 and 2 indicate the following observations:

- In most of these cases HYPER only generates a small fraction of the total refinement forest up to solution depth.
- The losses due to no duplicate checking are not dramatic, at least for the tabulated case of member/2. Also, as Table 2 shows, these losses are largely alleviated by the covers-alone heuristic.

Search in HYPER is guided by two factors: first, by the constraint that only complete hypotheses are retained in the search; second, by the evaluation function. In the cases of Table 1, the evaluation function guides the search rather well except for invariant/3.

One learning problem, not included in Fig. 1, where it seems HYPER did not perform satisfactorily, is the learning of quick-sort. In this problem, the evaluation function does not guide the search well because it does not discriminate between the hypotheses on the path to the target hypothesis from other hypotheses. The target hypothesis emerges suddenly „from nothing“. HYPER did induce a correct hypothesis for quick-sort with difference lists, but the definition of the learning problem (input-output modes) had to be defined in a way that was judged to be too unnatural assuming the user does not guess the target hypothesis sufficiently closely. Natural handling of such learning problems belongs to future work.

Table 1. Complexity of some learning problems and corresponding search statistics. "Total size" is the number of nodes up to Refine depth in the refinement forest defined by HYPER's refinement operator, taking into account the „covers-alone" heuristic.

Learning problem	Backgr. pred.	Pos. exam.	Neg. exam.	Refine depth	Hypos. refined	To be refined	All generated	Total size
member	0	3	3	5	20	16	85	1575
append	0	5	5	7	14	32	199	$> 10^9$
even + odd	0	6	5	6	23	32	107	22506
path	1	6	9	12	32	112	658	$> 10^{17}$
insort	2	5	4	6	142	301	1499	540021
arches	4	2	5	4	52	942	2208	3426108
invariant	2	6	5	3	123	2186	3612	18426

Table 2. Complexity of the search problem for inducing member/2. D is refinement depth, N is the number of nodes in refinement forest up to depth D with covers-alone heuristic, N(uniq) is the number of unique such hypotheses (i.e. after eliminating duplicates); N(all) is the number of all the nodes in refinement forest (no covers-alone heuristic), N(all,uniq) is the number of unique such hypotheses.

D	N	N(uniq)	N(all)	N(all,uniq)
1	3	3	6	6
2	13	13	40	31
3	50	50	248	131
4	274	207	1696	527
5	1575	895	12880	2151

4 Conclusions

The paper investigates the refinement of complete hypotheses. This was experimentally investigated by designing the ILP program HYPER which refines complete hypotheses, not just clauses. It does not employ a covering algorithm, but constructs a complete hypothesis „simultaneously". This alleviates problems with recursive definitions, specially with mutual recursion (when *both* mutually recursive clauses are needed for each of them to be found useful). The obvious worry with this approach is its increased combinatorial complexity in comparison with covering. The experimental results are possibly surprising in this respect as shown inTable 1. In most of the experiments in typical simple learning problems that involve recursion, HYPER's search heuristics cope well with the complexity of the hypothesis space.

The heuristics seem to be particularly effective in learning predicates with structured arguments (lists). On the other hand, the heuristics were not effective for invariant/4 whose arguments are not structured, and background predicates are arithmetic.

Some other useful properties of the HYPER approach are:

1. The program can start with an arbitrary initial hypothesis that can be refined to the target hypothesis. This is helpful in cases when the user's intuition allows start the search with an initial hypothesis closer to the target hypothesis.

2. The target predicate is not necessarily the one for which the examples are given. E.g., there may be examples for predicate p/1, while an initial hypothesis contains the clauses: q(X) and p(X) :- r(X,Y), q(Y).

3. The program always does a general to specific search. This monotonicity property allows to determine bounds on some properties of the reachable hypotheses. E.g. if H covers P positive examples and N negative examples, then all the hypotheses reachable through refinements will cover at most P and N examples respectively.

Acknowledgements

This work was partly supported by the Slovenian Ministry of Science and Technology, and the Commission of the European Union within the project ILP2. Part of this work was done during the author's visit at the AI Laboratory, School of Computer Sc. and Eng., University of New South Wales, Sydney, Australia, that also provided support for the visit. I would like to thank Marko Grobelnik, Uros Pompe and Claude Sammut for discussion.

References

1. Bratko, I., Grobelnik, M. (1993) Inductive learning applied to program construction and verification. *Proc. ILP Workshop 93*, Bled, Slovenia.
2. Grobelnik, M. (1992) Markus – an optimised model inference system. *Proc. ECAI Workshop on Logical Approaches to Machine Learning*, Vienna.
3. Muggleton, S. (1995) Inverse entailment and Progol. *New Generation Computing*, **13** (1995), 245-286.
4. Pompe, U. (1998) *Constraint Inductive Logic Programming*. Ph.D. Thesis, University of Ljubljana, Faculty of Computer and Info. Sc. (In Slovenian).
5. Quinlan, J.R. (1990) Learning logical definitions from relations. *Machine Learning*, **5** (1990), 239-266.