人　工　知　能 97－3
（1994．11．15）

例題からのプログラム合成における バイアスの緩和法

チョドゥリ ラハマン モフィズル，沼尾正行
東京工業大学情報理工学研究科計算工学専攻
Email: {rahman, numao}@cs.titech.ac.jp

例題と背景知識とから Prolog プログラムを合成するシステムがいくつか提案されているが，合成の各ケースごとに意味的，構文的に強力なバイアスをかける必要があり，広範囲のプログラムを対象とするのが困難であった．Prolog のプログラムでは，各変数への代入が 1 回に制限されており，計算の中間結果ごとに新しい変数が使用され，変数集合が大きくなりがちである．ところが，人間のプログラミングを調べると，各述語の実行時点において未参照の変数に注目して，未参照の変数の数が概ね最小になるように述語を選択していることが判明した．本論文では，そのようなプログラミング上の考え方を形式的に表現して，自動合成手法に導入することを提案し，他の手法との比較を行った．その結果，実験的に良好な結果を得ることができた．

Relaxing Bias on First Order Theory Formation

Chowdhury Rahman Mofizur and Masayuki Numao
Department of Computer Science
Tokyo Institute of Technology
2-12-1 O-okayama, Meguro 152, Japan
Email: {rahman, numao}@cs.titech.ac.jp

This paper presents a system for constructing first order Horn clause theories from examples and necessary background knowledge. The existing systems have adopted some form of strong semantic and syntactic bias to bound the intractable search space, but in doing so most of them have lost generality. Thus they behave well in specific fields, but fail in other situations. The implemented system ILPCS has incorporated some innovative natural semantic constraints which allows it to search in a restricted search space without a serious loss of generality. The detection of unnecessary redundant information together with limiting information explosion has enabled it as a natural system for learning first order theories. It has been verified experimentally that the set of learnable problems by ILPCS includes the set of benchmark problems learnable by the existing ILP systems with fewer example set and computational efforts.

1 Introduction

There has been enormous efforts in achieving general purpose ILP system which will be able to construct meaningful first order theory from teacher supplied examples. The main obstacle to this aim is to search an enormous intractable hypothesis space even when the language used to express the desired theory is very simple, like first order Horn clause expressions. Researchers have attempted to overcome this difficulty in many ways, mostly using some form of semantic and syntactic bias to constrain this search space. Some of these systems have even sought the help of an oracle to select the appropriate building block from a number of alternatives at each step of theory formation.

This paper presents a system for constructing first order Horn clause theories from examples and necessary background knowledge. The existing systems have adopted some form of strong semantic and syntactic bias to bound the intractable search space, but in doing so most of them have lost generality. Thus they behave well in specific fields, but fail in other situations. The implemented system ILPCS has incorporated some innovative natural semantic constraints which allows it to search in a restricted search space without a serious loss of generality. The detection of unnecessary redundant information together with limiting information explosion has enabled it as a natural system for learning first order theories. It has been verified experimentally that the set of learnable problems by ILPCS includes the set of benchmark problems learnable by the existing ILP systems with fewer example set and computational efforts.

2 Motivation

ILPCS borrows its idea from relational database concept. Learning a concept using first order Horn clause representation is merely a search through a number of background databases to find the consistency of the target relation. This is more evident when a system searches the hypothesis space in top-down manner from more general towards more specific concepts. Let us illustrate it using the following relational clause:

```
target(X,Y,Z):- backgr1(X,X1), backgr2(X1,Y,Z1), backgr3(Z1,Z).
```

We can view `backgr1`,`backgr2` and `backgr3` as relational databases with arity 2, 3 and 2 respectively. The learning task is to express the tuples {`X, Y, Z`} specified by the relation `target` in terms of the existing background databases. In other words, the expression to be learned, will tell us what databases to search using what attributes to find the consistency of the information expressed by the tuple {`X, Y, Z`}. The top-down approach is a search to find a link between the candidates of the target tuple (i.e. arguments of the target relation) using background relations. The variables which are introduced in the body of the clause thus represent linker or key attribute to search the next appropriate database. For example, in the above clause, attribute `X1` of relation `backgr1` coupled with `Y` act as a key to search relational database `backgr2`. Before that, `X` has been used to find an attribute `X1` using database `backgr1` which acts as a key attribute to search the next appropriate relation. Note that after the first literal, `X` has no role to play for the further clause development. Again after the second literal, attribute `X1` and `Y` have no function for the remaining clause. Thus it is apprehensible that after generating appropriate key attribute, some of the old attributes become redundant. Therefore, at points of clause development, some of the attributes become dead, the others remain alive to play a role in subsequent search. But unfortunately, the existing ILP systems did not attempt to get rid of this redundant attributes. In other words, existing systems consider all attributes equally important at every point of clause construction making the search space intractable. One of the innovative features of ILPCS, among others, is to detect the redundant attributes at appropriate points using some mechanism which will be described in the rest of the paper.

Another important point to consider is that if we view clause development is a kind of database search, then newly generated alive attributes together with old alive ones must be used to access the appropriate databases. As such, the total number of alive attributes must be bounded by a maximum limit which is dictated by the nature of the user supplied background predicates.

3 Overview of ILPCS

ILPCS learns from the following input:

- A set of positive and negative examples of the target predicate in the form of ground facts.
- Mode declarations declaring data types and input/output modes of variables in a literal.
- Background knowledge either in the form of first order Horn clause expressions or in extensional format.

Before proceeding further, we like to define and illustrate some terminologies used in the description of ILPCS.

Some Terminologies and their Explanations

The hypothesis search process involves discarding redundant variables and unpromising clauses. These variables and clauses can be defined as follows:

3.1 Dead Variables

We argued in the previous section that an ILP system can get rid of a sufficient number of redundant attributes from consideration at each point of clause development. Together with it, putting a limit on the amount of working information and the detection of unpromising clauses can enable an ILP system turn the intractable search space into tractable one. We have already explained in detail the source of redundant information using database concept. Let us now explain the phenomena from a programmer's point of view.

When a programmer begins writing a logic program, he or she transforms the input data (input state) to that of the output data (output state) using intermediate states (a set of antecedent literals). In doing this he or she introduces many new variables and also discards some variables which he or she will not use further in a clause development. For example, let us consider the following quicksort program:

```
sort([],[]).
sort([X|Xs],Ys):- partition(X,Xs,Ls,Gs),
                  sort(Ls,L1), sort(Gs,G1),
                  components(X,G1,G2), append(L1,G2,Ys).
```

Let a programmer P is trying to develop the second clause of this program. After adding the `partition` literal, P knows he or she will not use the variable `Xs` any more because `Ls` and `Gs` are now representing `Xs`. Again after adding `sort(Ls,L1)`, P knows he or she will not use `Ls` in his or her further development of the clause. From this example, it is observed that a programmer introduces many new variables and at the same time discards some variables already used in the partial clause. These variables which are discarded by the programmer from future use have been termed in ILPCS as dead variables. The characteristic property of a dead variable is that it does not discriminate the positive examples from the negative. As noted in the previous section, since the dead variables are redundant, they must not play a role for discrimination. In other words, since a dead variable will not take part in the remainder of the clause, the other variables are sufficient enough to discriminate positive examples from the negatives causing dead variable non-discriminating.

Now the question is how to detect these variables and how these variables help to reduce the search space. ILPCS uses two alternatives to detect dead variables:

- The user can supply the position of dead variables in the background literals.
- ILPCS can explicitly check the discriminating ability of the variables to find out the dead ones.

In current implementation ILPCS can use both because in real life situation the user may have knowledge about exactly which are the dead variables in some particular background literal whereas he or she may not have any knowledge of this sort for other literals as well. The user can specify background knowledge like `position (partition, [2])` to describe that if `partition` is used as a literal in a clause, then variable in argument position 2 is a dead variable. On the other hand the user may not supply such background knowledge and in that case ILPCS will explicitly detect that variable as a dead one. The procedure for detecting a dead variable explicitly is as follows:

ILPCS detects and stores dead variables literal by literal as the refinement process continues. Let the refinement procedure adds the literal `partition(X, Xs, Ls, Gs)` on the right hand side of the second clause of the quicksort program. ILPCS checks the set `X, Xs` for dead variables excluding the newly introduced variables `Ls,Gs`, because the new ones will be used as keys for further database search. ILPCS has now a set of positive and negative tuples consisting of the variables `X, Xs, Ys, Ls, Gs`. To check `X`, whether it is dead or not, ILPCS removes `X` from the tuple set and checks whether without `X`, the positive tuples are still different from the negative ones. If it is true, `X` is not necessary for discrimination and included in the dead variable set. The procedure continues with other variables in the same way. It is to be noted that after each round of refinement, the system has to check only a small set of variables for redundancy which occur in a newly added literal at the input argument positions.

If a clause which uses a dead variable in a subsequent literal after the point at which it has been marked dead, is obviously an unpromising clause. This sort of unpromising clause forms a huge hypothesis search space of no importance. By detecting and storing dead redundant variables, a system can easily get rid of this huge search space. The existing top-down ILP systems did not take into account of this kind of redundancy and thus could not take the benefit of escaping this huge search space.

3.2 Unpromising Clauses

3.2.1 Limiting the explosion of attributes

The set of dead variables may include variables from the target predicate as well as newly introduced variables in the antecedents. Therefore, there is a subset of the newly introduced variables which are out of date being members of the dead set. The remainder of the newly introduced variables which are not members of the dead set have been termed in ILPCS as alive new variables. It is to be noted that the set of alive new variables changes as the clause development proceeds. These alive new variables are merely used for searching appropriate background databases. Therefore, the arity of the background predicates must play a role to limit the total number of this kind of variables. It is evident that since the creation and destruction of new variables are carried out at the same time, the number of alive new variables must be bounded by some threshold value. Otherwise a clause will have infinite length. ILPCS has used the maximum between the following two values:

- The maximum number of input arity of the given background knowledge.
- The maximum number of output arity of the given background knowledge.

If any clause violates this threshold value, it is discarded from the beam being unpromising in nature. This natural choice of unpromising clauses constitute a huge search space which ILPCS has been able to escape by adopting this heuristic. It should be noted that the adoption of threshold value has enabled ILPCS to decrease the tendency of inserting the same determinate literal more than once which merely increases the number of alive new variables without increasing gain.

3.2.2 Normalized Gain

FOIL[7] has used the information gain heuristic for hill climbing search through the hypothesis search space. But this heuristic measure is not sufficient enough to find all concepts expressible in first order Horn clause logic. Non-discriminating literals with zero or very small gain poses a problem for the system to incorporating them in the clause. The concept of determinant literal with post-processing has enabled it recover from this problem. We have used information gain measure in a different form not for selecting the appropriate literal, but for discarding unpromising clauses and constraining search space.

Gain of a literal L_i is defined as follows:

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

where T_i^{++} is the number of positive tuples in T_i that have extensions in T_{i+1}. $I(T_i)$ and $I(T_{i+1})$ are the information required for signalling that a tuple in a training set T_i and T_{i+1} are one of the positive kind, respectively. Gain is negative if the positive tuples are less concentrated after adding a literal and small if either the concentrations are similar or few positive tuples satisfy the literal. The usefulness of a literal is measured by its normalized gain $(I(T_i) - I(T_{i+1}))$, rather than gain, since the value of the normalized gain, does not depend on the size of the training examples. The value of the gain can be large due to a large training set even though the literal poorly discriminates the examples. We have measured the normalized gain difference after each literal added by the refinement operator. If the addition of a literal causes the partial clause to take negative value on absolute gain difference, we have discarded that clause from our search space. Thus, using information gain measure loosely, not for literal selection, but for the detection of unpromising clauses, has enabled ILPCS to behave generally.

Let us now describe the ILPCS algorithm in details. ILPCS uses top-down induction like FOIL using beam search. It uses the refinement operators in MIS[8] to specialize a clause by: (i) instantiating a head variable to a function, (ii) unifying variables, and (iii) adding a background predicate to the body. ILPCS randomly selects a seed example and begin to specialize the most general term MGT of this seed example. After each round of refinement, there are two tasks to be performed by ILPCS:

1. Discard the unpromising clauses from the search space:
    (a) Detect the clauses which have added a new literal with dead variables. Discard these clauses from the generated beam.
    (b) Measure the absolute gain differences for the refined clauses. Discard the clauses which take on negative values for absolute gain difference.
2. Check the remaining clauses for consistency. If one found, remove the +ve examples covered by the clause and start from another seed example if there remains any +ve examples to be covered. Otherwise
    (a) check each clause for the number of alive new variables. If this number exceeds the predetermined threshold number for some clause, discard that clause from the beam.
    (b) Detect the newly formed dead variables in the newly added literal if any, for each clause. Update the dead variable set for each clause.

    Arrange the clauses in order of their desirability. Take N (Beam width) clauses out of the sorted clauses and repeat the refinement process.

Next we will discuss the ordering of the clauses as applied in ILPCS. The ordering of the clauses depends on the number of alive new variables, N1 and number of newly incorporated dead variables, N2. ILPCS has measured the desirability of a clause as directly proportional to N1 (note that N1 is bounded by a threshold value) and inversely proportional to N2. This implies that ILPCS gives credit to clauses which produce more alive new variables bounded

```
Procedure ILPCS
Input: A set of positive and negative examples of target predicate
A database of background knowledge either in extensional,
intensional or in mixed format
Output: A set of complete and consistent clauses
begin
N :- max of {max input arity, max output arity} of background literals
  while there are some positive examples uncovered do begin
    Randomly choose a seed example E from positive examples
    Creat the most general term of E, call it MGT
    Q := {MGT}; R := {}; DeadSet := {}
      while all elements of Q cover any -ve example do begin
        for each element of Q do
          Compute the refinements of the element that cover E using
          the refinements operators, and add them to R
        endfor
        for each clause in R do
          Compute the absolute gain difference and if negative
          discard the clause from search space
          Discard a clause if added new literal contains
          member variables from the DeadSet otherwise
          Update DeadSet and compute the number N1 of alive new variables
          If N1 > N, discard the clause else include it in set R
        endfor
        Compute the desirability of each clause in R and order
        them according to the desirability
        Take first K (beam width) clauses as a set Q for refinement
      endwhile
  endwhile
end
```

Figure 1: ILPCS Algorithm

by the threshold value and gives discredit to a clause which makes its variables redundant very rapidly. ILPCS adds the desirability of clauses cumulatively as the refining process goes on.

4 Initial Experimental Results

We have tested the learning ability of ILPCS in a number of examples. It includes the set of logic programs learnable by the existing ILP systems. Note that most of the existing systems can learn a subset of the problem learned by ILPCS. Below is the listing of representative set of logic programs successfully learned by ILPCS:

Quick Sort Program:

```
  sort([],[]).
  sort([X|Xs],Ys):- partition(X,Xs,Ls,Gs),sort(Ls,L1),sort(Gs,G1),
                     components(X,G1,G2), append(L1,G2,Ys).
```

List Reverse Program:

```
  reverse([],[]).
  reverse([X|Xs],Ys):- reverse(Xs,Zs), concat(Zs,X,Ys).
```

Can-reach Program:

```
  can-reach(X,Y):- link-to(X,Y).
  can-reach(X,Y):- link-to(X,Z), can-reach(Z,Y).
```

N-queen Program:

```
  nqueen([],Qs,Qs).
  nqueen(UnpQs,SafeQs,Qs):- select(Q,UnpQs,UnpQs1),component(Q,SafeQs,SQs1),
                             nqueen(UnpQs1,SQ1,Qs),not-attack(Q,SafeQs).
```

Multiply:

```
  multiply(0,X,0):- zero(0).
  multiply(1,X,X):- one(1).
  multiply(X,Y,Z):- dec(X,W), multiply(W,Y,Z1), add(Z1,Y,Z).
```

In the above listing, both non-discriminating and non-determinant literals are present. For example, the n-queens program is a generate and test program containing the non-determinant literal `select` which is not learnable by GOLEM[6]. Again most of the literals in quicksort program are non-discriminating in nature which has been learned efficiently by ILPCS without applying any special heuristic as has been done in FOIL. Note that ILPCS learns all the alternative definition of a clause if appropriate for that particular clause. For example it has learned more than one definition for some clauses in the the quicksort and nqueens programs which have not been shown due to the lack of space.

Problem	Training Set	Nodes Generated	Nodes Visited	Time used in ms
Experiments with Smaller Example Set				
QuickSort	8 +ve Exs 7 -ve Exs	161	29	4710
Reverse	5 +ve Exs 7 -ve Exs	19	9	209
Can-reach	7 +ve Exs 17 -ve Exs	9	9	170
N-queens	6 +ve Exs 12 -ve Exs	54	14	1089
Multiply	12 +ve Exs 11 -ve Exs	56	30	1589
Experiments with Larger Example Set				
QuickSort	23 +ve Exs 17 -ve Exs	178	27	10129
Reverse	16 +ve Exs 256 -ve Exs	19	9	2859
Can-reach	19 +ve Exs 62 -ve Exs	14	9	1120
N-queens	239 +ve Exs 73 -ve Exs	68	14	34519
Multiply	12 +ve Exs 11 -ve Exs	56	30	2910

Table 1. Experimental Results

In Table 1, we have shown the learning performance of ILPCS in the domain of logic programs. It is to be noted that ILPCS can learn from a few number of selective examples without the interaction of an oracle. We do not know any other existing top-down learning system which can do the same without the interaction of user. The positive point is that ILPCS can learn the same programs from a relative larger set of random examples. The performance difference is that ILPCS now takes more time to handle a larger set of examples as shown in Table 1. since ILPCS does not compute any heuristic parameter for hill climbing search through the hypothesis space, it is natural that it will be able to induce meaningful theory with much fewer examples.

5 Conclusion

The algorithm underlying the ILPCS system is different in nature from the other existing systems and is more related to inducing a program like a human programmer. ILPCS evolves in a natural way using some innovative idea which can be applied fruitfully to any existing ILP system. Its original idea is to search in a limited search space using some natural weak semantic bias. Since most of the existing ILP system either employ heuristics or seek the interaction from the user to reduce the search space, they are smart in particular area of application. The adoption of strong bias has limited existing ILP systems in their application whereas ILPCS, influenced least by such a strong bias, naturally is able to learn a larger family of first order Horn clause theories. We believe the idea reported in this paper will motivate the researchers to update existing algorithms and to promote new approaches in learning first order theories. Our future research direction is to enhance ILPCS system to withstand noisy data. Since ILPCS works not only with the literal level, but also with the arguments within a literal, we think it will be less affected with overfitting noisy data. However, it needs experimentation to verify this conjecture.

References

[1] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Procedings of the 13th IJCAI*, 1993.

[2] B. Kijsirikul, M. Numao, and M. Shimura. Efficient learning of logic programs with non-determinate, non-discriminating literals. In *8th Intl. Workshop on Machine Learning*, 1991.

[3] B. Kijsirikul, M. Numao, and M. Shimura. Discrimination-based constructive induction. In *AAAI*, 1992.

[4] Chowdhury Rahman Mofizur and Masayuki Numao. Constructive induction for recursive programs. In *Procedings of the 4th International conference on Analogical and Inductive Inference, Lecture Note series, vol. 872, Springer-Verlag*, 1994.

[5] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *5th Intl. Workshop on Machine Learning*, 1988.

[6] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Procedings of the 1st International workshop on Algorithmic Learning Theory*, 1990.

[7] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[8] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.

[9] R. Wirth and P. O'Rorke. *Constraints for Predicate Invention.* Inductive Logic Programming, Academic Press, 1992.