An Adaptive Inductive Logic Programming System using Genetic Programming

Man Leung Wong
Department of Computing and Decision Sciences
Lingnan University, Tuen Mun
Hong Kong
mlwong@ln.edu.hk

Kwong Sak Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong
ksleung@cse.cuhk.edu.hk

Abstract

Recently, there have been increasing interests in Inductive Logic Programming (ILP) systems. But existing ILP systems cannot improve themselves automatically. This paper describes an Adaptive Inductive Logic Programming (Adaptive ILP) system that evolves during learning. An adaptive ILP system is composed of an external interface, a biases base, a knowledge base of background knowledge, an example database, an empirical ILP learner, a meta-level learner, and a learning controller. A preliminary adaptive ILP system has been implemented. In this implementation, the empirical ILP learner performs top-down search in the hypothesis space defined by the concept description language, the language bias, and the background knowledge. The search is directed by search biases which can be induced and refined by genetic programming (Koza 1992).

It has been demonstrated that the adaptive ILP system performs better than FOIL, a famous ILP system (Quinlan 1990), in inducing logic programs from perfect or noisy training examples. The experimentation illustrates the benefit of an adaptive ILP system over existing ILP systems. The result implies that the search bias induced by genetic programming (GP) is better than that of FOIL, which is designed by a top researcher in the field. Consequently, GP is a promising technique for implementing a meta-level learning system. The result is very encouraging as it suggests that the process of natural selection and evolution can successfully evolve a high performance ILP system.

1 INTRODUCTION

Recently, there has been increasing interest in systems that induce first-order logic programs. In this formalism, domain knowledge represented in the forms of first-order relations can be used in the induced programs. The task of inducing a logic program can be formulated as a search problem (Mitchell 1982) in a hypotheses space of logic programs. Various approaches (Quinlan 1990; Muggleton and Feng 1990) differ mainly in the search strategy and the heuristics used to guide the search. The search

space is extremely large, so strong heuristics are required to manage the problem. Most systems are based on a greedy search strategy. They generate a sequence of logic programs from general to specific (or from specific to general) until a consistent program is found. Each program in the sequence is obtained by specializing (or generalizing) the previous one. For example, FOIL (Quinlan 1990) applies a hill climbing search strategy guide by an information-gain heuristic to search programs from general to specific. But these strategies and heuristics are not always applicable because the systems may become trapped in local maxima. In order to overcome this problem, non greedy strategies should be adopted.

Genetic algorithms (GAs) are alternative search strategies which perform an implicitly parallel search (Goldberg 1989). Genetic algorithms perform both exploitation of the most promising solutions and an exploration of the search space. Wong and Leung have developed a system called Genetic Logic Programming System (GLPS) that employs GAs to induce logic programs (Wong and Leung 1994a, 1994b).

Moreover, existing ILP systems cannot improve themselves automatically and this paper describes an Adaptive Inductive Logic Programming (Adaptive ILP) system that can evolve during learning based on genetic programming (Koza 1992). The next section formulates the definitions of inductive concept learning and adaptive inductive logic programming. Section three describes a generic top-down ILP algorithm. The fourth section presents a meta-level learner that induces search bias. Section five delineates the experimentation and some evaluations of the system followed by a conclusion.

2 INDUCTIVE CONCEPT LEARNING AND ADAPTIVE INDUCTIVE LOGIC PROGRAMMING

The goal of machine learning is to develop techniques and tools for building intelligent learning machines. Machine learning paradigms include inductive, deductive, genetic-based, and connectionist learning. Multi-strategy learning integrates several learning paradigms. This section focuses on supervised inductive concept learning. If **U** is a universal set of observations, a concept **C** is formalized as a subset of observations in **U**. Inductive concept learning finds descriptions for various target concepts from positive and negative training instances of these concepts.

In machine learning, formal languages for describing observations and concepts are called object and concept description languages respectively. Typically, object description languages are attribute-value pair descriptions and first-order languages of Horn clauses. Concepts can be described extensionally or intensionally. A concept is described extensionally by listing the descriptions of all of its instances (observations). Thus extensional concepts are represented in the object description language. On the other hand, intensional concepts are expressed in a separate concept description language that permits compact and concise concept descriptions. Typical concept description languages are decision trees, decision lists, production rules, and first-order logic.

Inductive concept learning can be viewed as searching the space of hypothesis descriptions. A bias is a mechanism employed by a learning system to constrain the search for target hypotheses. A search bias

determines how to conduct the search in the hypothesis space while a language bias determines the size and structure of the hypothesis space.

A strong search bias, such as the hill-climbing search strategy, employs existing knowledge about the size and structure of the hypothesis space to exploit promising solutions of the space, thus it can find the target concept quickly. But it may trap the system in a local maximum. A weak search bias, such as depth-first and breath-first search, explores the space completely; the learner is guaranteed to find the target concept that can be represented by the concept description language. Nevertheless, a weak bias is very inefficient. In other words, the search bias introduces the efficiency/completeness tradeoff into a learning system.

A strong language bias defines a less expressive description language such as the propositional logic. The hypothesis space created by the bias is comparatively smaller and the learning can be performed more efficiently. But the learner may fail to find the target concept which is not contained in the small hypothesis space. A weak bias defines a larger space and thus the target concept is more likely to be expressible in the space. The disadvantage is that the learner is less efficient. The language bias introduces the efficiency/expressiveness tradeoff into a learning system.

Background knowledge **B** is declarative prior knowledge that can be used by either the search bias to direct the search more efficiently, or the language bias to express the hypothesis space in a more natural and concise way. Background knowledge plays an important role in relational concept learning. Relational concept learning induces a new relation for the target concept (i.e., the target predicate) from training examples and known relations from the background knowledge. An inductive Logic Programming (ILP) system is a relational concept learner. The training examples, the hypothesis space and the background knowledge are represented in first order Horn clause languages (Muggleton and Feng 1990). Tradeoffs between expressiveness and efficiency are introduced by some additional restrictions on the three languages. The background knowledge **B** provides definitions of known predicates q_i which can be used in the definition of the target predicate p. It also provides additional information to ease the learning. The information includes argument types, symmetry of predicates in pairs of arguments, input/output modes, rule models, predicate sets, parametrized languages, integrity constraints, determinations and any knowledge that can modify the operation of the search and language biases.

An Adaptive Inductive Learning Programming (Adaptive ILP) system is an ILP system that can improve itself on the learning capability. It maintains various sets of background knowledge and biases. It improves itself by modifying its biases and background knowledge. A hypothesis space for learning is defined through the concept description language, the language bias and the background knowledge. Therefore, by changing the language bias and the background knowledge, the size and structure of the hypothesis space can be modified accordingly. The search strategy and heuristics are changed if the system's search biases are modified. Here, we formulate the task of an Adaptive ILP system as follows:

Given:

-A set **E** of positive $\mathbf{E}^+$ and negative $\mathbf{E}^-$ training examples of the target predicate p.
-A concept description language **L**
-A set of learning biases **BIASES**
-A set of various background knowledge **BKs**

Find:

-A modified set of learning biases $\mathbf{BIASES}'$
-A modified set of background knowledge $\mathbf{BKs}'$
-A concept definition **H** for the target predicate p expressible in **L** such that **H** is complete and consistent with respect to (w.r.t.) the training examples **E** and a background knowledge $\mathbf{B}'$ in $\mathbf{BKs}'$

H is complete if every positive example e^+ in $\mathbf{E}^+$ is covered by **H** w.r.t. the background knowledge **B**. i.e. $\mathbf{B} \cup \mathbf{H} \models e^+$

H is consistent if no negative example e^- in $\mathbf{E}^-$ is covered by **H** w.r.t. the background knowledge **B**. i.e. $\mathbf{B} \cup \mathbf{H} \not\models e^-$

The logical organization of our adaptive ILP system is depicted in Figure 1. Its components are introduced as follows:

(1) *External interface*: It provides a user-friendly interface between the system and users. It accepts training examples, a set **BKs** of background knowledge, and a set **BIASES** of biases and transfers them through the learning controller to the *example database*, *BKbase* and *biases base* respectively. The interface also provides commands for users to query about the results of an adaptive learning task and to directly control the operations of the learning controller.

(2) *Biases base*: It is a knowledge base that stores all learning biases.

(3) *BKbase*: It stores various background learning knowledge that can be used in inductive learning. Background knowledge can be retrieved, added, deleted and modified through the interface of BKbase.

(4) *Examples database*: It stores the training examples.

(5) *Empirical ILP learner*: It induces a logic program from the training examples, given a concept description language, a specific background knowledge, a search bias and a language bias. A search of the hypothesis space can be performed bottom-up or top-down. Bottom-up techniques start from the training examples and search the space by employing various generalization operators. Top-down techniques start from the most general concept descriptions, and search the space by using various specialization operators. Top-down techniques are better suited for learning from imperfect examples because a large number of data are available in every specialization step and the system can employ various statistical techniques to decide how to perform the specialization. Moreover, top-down search can easily be guided by the

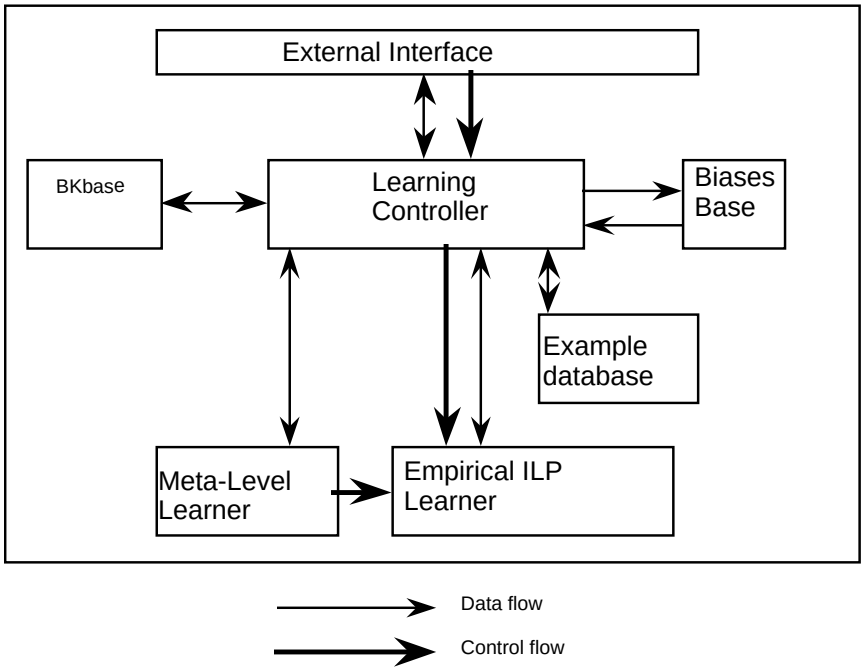

Figure 1. The logical organization of an adaptive ILP system

search bias. In section 3, a generic top-down ILP algorithm will be described.

(6) *Meta-level learner*: It learns search biases, language biases, and background knowledge. Search and language biases can be represented declaratively or procedurally. Section 4 describes a meta-level learner implemented in genetic programming (Koza 1992) to induce procedural biases. The induced definition of the current target predicate is stored to facilitate the learning of higher level predicates, thus an adaptive ILP system is a closed learning system. It can also learn other meta-knowledge such as the conditions under which various learning biases and background knowledge can be employed. Since the meta-level learner performs a variety of learning tasks, it is implemented as a multi-strategy learning system.

(7) *Learning controller*: It is a knowledge-based system that controls the empirical ILP learner and the meta-level learner. The knowledge used by the learning controller can be updated by the meta-level learner.

3 A GENERIC TOP-DOWN ILP ALGORITHM

This section presents a generic top-down ILP algorithm. The algorithm is depicted in Figure 2. The algorithm consists of three steps. In the pre-processing step, missing argument values in training examples are handled by assigning default or random values to them. A training example will be removed if it has too many missing values. If there are no or inadequate negative examples in the training set, they can be generated.

Different ways of creating negative examples have been proposed (Lavrac and Dzeroski 1994).

The second step performs the construction of a program. This step employs four local variables: $\mathbf{E}_{current}$ (Current training examples set), $\mathbf{E}'_{current}$ (Updated training examples set), **P** (Current program) and $\mathbf{P}'$ (Modified program). The main component of this step is the covering loop which implements Michalski's covering algorithm (Michalski et al. 1986). The covering loop construct a program by iteratively executing the following sub-steps:

(a) Construct a clause that covers some positive examples in $\mathbf{E}_{current}$.

(b) Append the clause to the current program **P** and generate a modified program $\mathbf{P}'$.

(c) Remove all positive examples from $\mathbf{E}_{current}$ which are covered by $\mathbf{P}'$ w.r.t. the background knowledge **B**.

The covering loop terminates if the terminating conditions are satisfied. A typical condition is that either all positive examples are covered or no more improvement can be achieved by searching for a new clause. The final step attempts to improve the accuracy of the program induced when classifying unseen examples and to simplify the program.

The covering loop calls the 'Clause-Construct' function which is the core of the generic algorithm. The function constructs a clause $C_n = T \leftarrow l_1, l_2, ..., l_n$ starting from the most general clause $C_0 = T \leftarrow$ with an empty body. A sequence of clauses $C_0, C_1, C_2, C_3, ... C_n$ are generated by a number of specialization steps. At each step, the current clause $C_i = T \leftarrow l_1, l_2, ..., l_i$ is refined by appending a specific literal l_j to its body. A literal l_j is constructed from the background knowledge **B** restricted by the concept description language **L** and language bias $BIAS_{lang}$. The language may limit l_j to be function-free while $BIAS_{lang}$ may prevent new variable to be introduced in l_j. The aim of the procedure is to find a clause which covers most positive examples while excludes all or most negative examples. In a hill-climbing search, the procedure keeps the current best clause and refines it using the estimated best specialization at each step, until the stopping condition is satisfied. A hill-climbing 'Clause-Construct' algorithm is presented in Figure 3.

```
Input:
    Training examples E
    Concept description language L
    Search bias BIAS_search and Language bias BIAS_lang
    background knowledge B
    Target concept T

Output:
    A program P which contains a set of program clauses. Each  clause
    C ∈ L.

Function ILP(E, L, BIAS_search, BIAS_lang, B, T)

(1) Pre-processing of the training examples E and producing a
    modified set of examples E': E' := Preprocessing(E).

(2) Let E_current := E';
    Let P := {};
    Repeat
        -Let C := T ←;
        -Find a specialization C' of C. This step constructs a
         clause C' from C by calling Clause-Construct(C,
         E_current, B, L,  BIAS_search, BIAS_lang);
        -If a specialization can be found
            -Add C' to P to produce a new program P'. i.e.
             P' := P ∪ {C'};
            -Remove all positive examples covered by P' from
             E_current to get an updated training set E'
             E'_current := E_current - { positive examples in
             E_current covered by P' w.r.t. the background
             knowledge B};
            -Let E_current := E'_current;
            -Let P := P'
         Else
            -Set the flag No-More-Improvement to true;
    Until
        The Covering termination criterion is satisfied. i.e.
        covering-termination(P, No-More-Improvement, E_current, B)
        returns true;

(3) Post-processing the program P and producing P'. i.e.
    P' := Post-processing(P);
    Return(P');
```

Figure 2. A generic top-down ILP algorithm

The 'Clause-Construct' function calls the 'Find-Extension' function to find the extension $\mathbf{E}_i$ of the current training examples given the partially developed clause $C_i = T(X_1, X_2, ..., X_n) \leftarrow l_1, l_2, ..., l_i$ and the background knowledge **B**. Each training example $<x_1, x_2, ...x_n>$ is a n-tuple where x_i, $1 \leq i \leq n$, are some constants. To find the extension, the function initializes a clause $C_0 = T(X_1, X_2, ..., X_n)$, then the literal l_1 is added to the body of C_0 to produce a new clause C_1. The literal l_1 is either of the form $X_j = X_k$, $X_j \neq X_k$, $p_m(Y_1, Y_2, ...,Ys_m)$ or not $p_m(Y_1, Y_2, ...,Ys_m)$.

```
Input:
    An initial clause C = T←
    The current training examples E_current
    Background knowledge B
    Concept description language L
    Search bias BIAS_search and language bias BIAS_lang

Output:
    A clause that covers some positive examples in E_current while
    excludes all or most negatives examples in E_current

Function Clause-Construct(C, E_current, B, L, BIAS_search,
                          BIAS_lang)

There is a scoring function stored in BIAS_search, save this
function to scoring;

Repeat
    -Set BEST to a bad literal such as X = X where X is a
     variable appearing in the head of the clause;
    -Set Best-score to 0;
    -Find the extension E_i of E_current using the clause C w.r.t.
     B. i.e. E_i := Find-Extension(C, E_current, B);
    -Let n_i^+ be the number of positive tuples in E_i;
    -Let n_i^- be the number of negative tuples in E_i;
    -Current-information := -log2(n_i^+ / (n_i^+ + n_i^-));
    -For all literal l from B that satisfy the constraints
     imposed by the language L and bias BIAS_lang
        -Set C' = C ∪ {l};
        -Find the extension E_i+1 of E_current using the clause C'
         i.e. E_i+1 := Find-Extension(C', E_current, B);
        -Let n_i+1^+ be the number of positive tuples in E_i+1;
        -Let n_i+1^- be the number of negative tuples in E_i+1;
        -Let the number of positive tuples in E_i that have been
         represented by one or more tuples in E_i+1 be n_i^++ ;
        -Find the score of the literal l by using the scoring
         function i.e. literal-score := scoring (n_i^++, n_i+1^+, n_i+1^-,
         Current-information);
        -If literal-score > Best-score then
            -BEST := l;
            -Best-score := literal-score;
    -If BEST == X=X then
        -No-More-Improvement := true;
     Else
        -Append BEST to the body of C;
Until Clause-Termination(C, No-More-Improvement, E_current, B)
      is true;

Post-processing the clause C to find an improvement i.e.
C' := Find-Improvement(C);

If Acceptable(C')
    -Return(C');
Else
    -Return(No-Specialization-Can-Be-Found);
```

Figure 3. A hill-climbing 'Clause-Construct' algorithm

If the literal contains one or more new variables {NewY$_1$, NewY$_2$, ..., NewY$_k$}, the arity of each tuple in the generated training set $\mathbf{E}_1$ increases to (n + k). $\mathbf{E}_1$ can be found by performing a natural join of $\mathbf{E}_{\text{current}}$ with

the relation corresponding to literal l_1. The process is repeated for literals l_2, l_3, ...l_i until the extension $\mathbf{E}_i$ is found.

The most important component of the hill-climbing 'Clause-Construct' algorithm is the 'scoring' function that estimates the performance of each literal. An accurate estimation directs the search towards the global maxima while a misleading one traps the system into local-maxima. By providing different 'scoring' functions to the generic ILP algorithm, various learning algorithms can be generated. The performances of a good and a bad learners can be significant different as shown in section 4.

4 INDUCING PROCEDURAL SEARCH BIAS

In this section, genetic programming (Koza 1992) is used in the meta-level learner to induce procedural search bias (i.e. the 'scoring' function). GAs (Holland 1975) have proven successful in finding optimal points in a search space for a wide variety of problems (Goldberg 1989). Genetic programming (GP) extends traditional GAs to learn a program composed of functions. Koza (1992) demonstrates that populations of computational functions can be genetically bred to solve a variety of problems in a wide variety of fields.

Primitive Definitions

In order to employ GP to induce the 'scoring' function, a list of terminals and primitive functions sufficient to solve the problem must be defined. The terminal set **T** includes the formal parameters of the 'scoring' function and the ephemeral random floating point constant $\mathbb{R}$, i.e.,

$$\mathbf{T} = \{n_{i+1}^{+}, n_{i+1}^{-}, n_{i}^{++}, \text{current-information}, \mathbb{R}\}$$

With reference to the algorithms in figures 2 and 3, assume that $\mathbf{E}_i$ is the extension of current training examples $\mathbf{E}_{\text{current}}$ by current clause C_i, n_i^+ and n_i^- are respectively the number of positive and negative tuples in $\mathbf{E}_i$. $\mathbf{E}_i$ can be extended by using the literal l to $\mathbf{E}_i{+}1$. n_{i+1}^+ and n_{i+1}^- are respectively the number of positive and negative tuples in $\mathbf{E}_i{+}1$. n_i^{++} is the number of positive tuples in $\mathbf{E}_i$ that have been represented by one or more tuples in $\mathbf{E}_i{+}1$. Current-information is defined as $-\log_2(n_i^+ / (n_i^+ + n_i^-))$. The ephemeral random floating point constant $\mathbb{R}$ takes on a different random floating point value between -10.0 and +10.0 whenever it appears in the initial population.

The function set **F** includes the arithmetic functions +, -, *, %, protected-log and Info, their arities are respectively 2, 2, 2, 2, 1 and 2, i.e.

$$\mathbf{F} = \{+/2, -/2, */2, \%/2, \text{protected-log}/1, \text{Info}/2\}$$

Typically, the protected division % and the protected logarithm protected-log return the quotient and logarithm respectively. However, if division by zero is attempted, % returns 1.0. If logarithmic of zero is

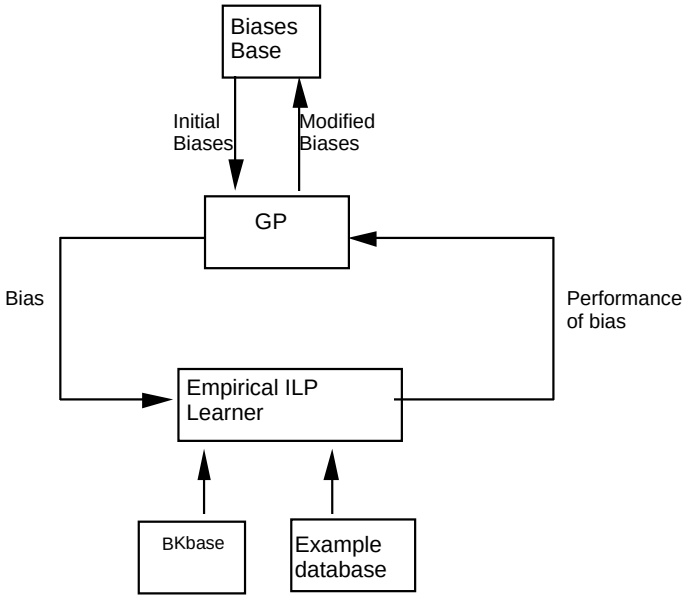

Figure 4. The evolution process of the adaptive ILP system

evaluated, protected-log returns 0.0. The basic function Info calculates $-\log_2(X/(X+Y))$ given X and Y as inputs.

The evolution process

The evolution process of the Adaptive ILP system is depicted in figure 4. Firstly, the Biases base is initialized with a population of different 'scoring' functions generated randomly using the primitives in **T** and **F**. To estimate the fitness of a specific 'scoring' function, it is combined with the generic top-down ILP learner to produce a specific ILP learner. The performance of this ILP learner is then evaluated by using a fitness function. This measure is assigned as the fitness of the specific 'scoring' function. GP employs crossover, selection, mutation, and other genetic operators to generate potentially better functions. The modified functions are stored in the Biases base and the whole evolution process iterates until the best function is found or no computational resource is available.

The experimentation setup

In this paper, learning curves are frequently used to estimate the performances of various learning systems. The example space is divided randomly into disjoint training and testing sets. The learner is trained on progressively larger portions of the training set and the performance of the induced logic program is estimated on the disjoint testing set. This process of dividing, training and testing is repeated for 20 trials and the results are averaged to generate a learning curve.

aaAs a running example, we use a traditional problem discussed in the literature (Muggleton and Feng 1990). In the problem of learning the list predicate *member*, the data consist of all lists of lengths 0 to 3 defined over three constants. The background knowledge **B** contains definitions of list construction predicates: *null* which holds for an empty list and *component*

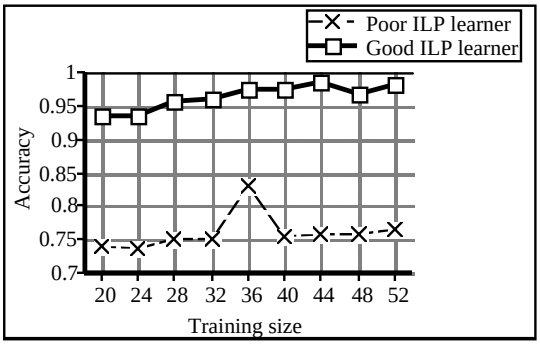

Figure 5. The learning curves of good and poor ILP learner

which decomposes a list into its head and tail. The example space contains 75 positive and 45 negative examples. The training sets contain 20 to 52 examples, one-half of each training set is positive examples. The testing set consists of 45 positive and 15 negative examples.

Fitness calculation

Adjusted and normalized fitness are used as in Koza (1992). They are calculated from the raw fitness which is estimated by the fitness function. Various fitness functions have been tried and two of them are described here. The impact of fitness function on the generality of the evolved function is also demonstrated. The problem domain of learning the member predicate is used here.

For the first fitness function, a random set of 24 positive and 21 negative examples is used. A specific 'scoring' function is combined with the generic top-down ILP learner to produce a specific ILP learner called GP-ILP hereafter. GP-ILP induces a logic program using the random example set. The quality of the induced logic program is evaluated by counting the total number of misclassified examples from the same training set. This measure is used as the raw fitness of the specific 'scoring' function. Using this fitness function, only poor 'scoring' functions have been evolved. The learning curve of a poor learner is depicted in figure 5.

For the second fitness function, the raw fitness is developed in several steps. At the beginning of each generation, four instances of the learning task are created randomly from the member domain. Each learning task has a training and a disjoint testing data. The training set contains 20 positive and 20 negative examples. For each learning task, a specific GP-ILP induces a logic program from the training set and the logic program is evaluated by counting the number of misclassified examples from the testing set. The performance of the GP-ILP is the sum of numbers of misclassified examples for all learning tasks. This measure is then used as the raw fitness of the corresponding 'scoring' function. This fitness function can force the evolution of good 'scoring' functions. The learning curve of a good learner is shown in figure 5.

5 EXPERIMENTATION AND EVALUATIONS

This section compares the performance of the adaptive ILP system with that of FOIL which is a famous ILP system (Quinlan 1990). Standard learning tasks in the literature are used in these experiments (Quinlan 1990; Muggleton and Feng 1990).

The member predicate

The learning curves for this problem are depicted in Figure 6. It is interested to find that the adaptive ILP system has higher accuracy than FOIL. The difference is significant at 95% confidence interval when the training size is less than 36.

The member predicate in a noisy environment

Difference amount of noise is introduced into the training examples in order to study the performances of both systems in learning programs in noisy environment. To introduce n% of noise into the examples, n% positive examples are labeled as negative ones while n% negative examples are labeled as positive ones. In this experiment, the percentages of introduced noise are 10% (0.1) and 40% (0.4). Their learning curves are summarized in figure 7. The adaptive ILP system performs better than FOIL at all noise level.

The multiply predicate

In the problem of learning the arithmetic predicate *multiply* (Muggletion and Feng 1990), the data contain integers in the range from zero to ten. The background knowledge is composed of definitions for arithmetic predicates *plus*, *decrement*, *zero* and *one*. The example space has 73 positive and 1258 negative examples respectively. The training sets consist of 400 to 500 examples, one-tenth of each training set is positive and the remainder is negative. The learning curves for multiply are presented in Figure 8. The Adaptive ILP system performs better than FOIL when the size of training set is less than 460. The difference is significant at 99.5% confidence interval.

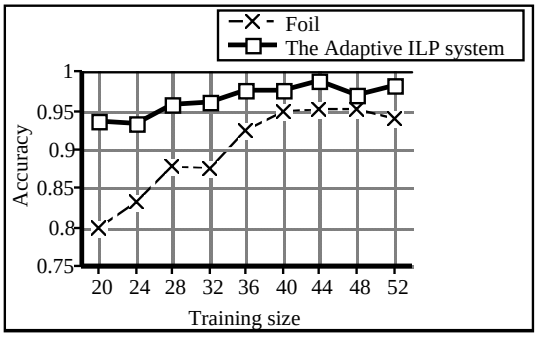

Figure 6. Learning curves for the member problem

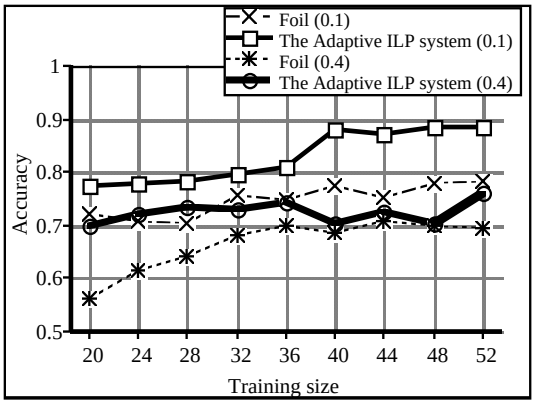

Figure 7. Learning curves for the member problem in a noisy environment

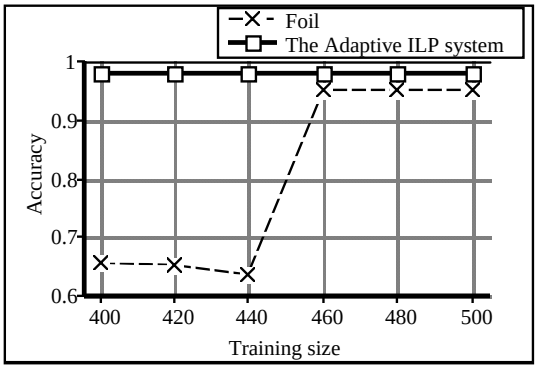

Figure 8. Learning curves for the multiply problem

The uncle predicate

Another traditional testbed for relational learners is the domain of family relationships (Quinlan 1990). In this experiment, the *uncle* predicate is induced and the background predicates are *parent*, *sibling*, *married*, *male* and *female*. The learning curves are presented in Figure 9.

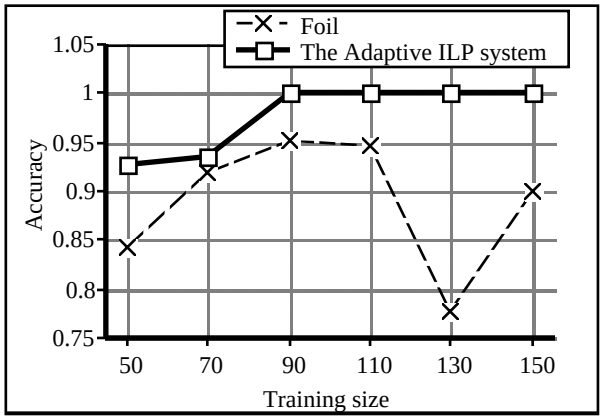

Figure 9. Learning curves for the uncle problem

6 CONCLUSION

In this paper, we formulate an Adaptive Inductive Logic Program system which is composed of an external interface, a biases base, a knowledge base of background knowledge, an example database, an empirical ILP learner, a meta-level learner and a learning controller. A preliminary implementation of an adaptive ILP system has been completed. In the implementation, the empirical ILP learner performs top-down search in the hypothesis space defined by the concept description language, the language bias and the background knowledge. The search is directed by search biases which can be induced and refined by genetic programming.

It has been demonstrated that the induced bias is better than that of FOIL on many standard learning tasks. From these experiments, it can be concluded that the Adaptive ILP system has superior learning ability compared to FOIL. Since they are different in their search biases only, the result implies that the search bias induced by GP is better than that of FOIL for the learning problems. This result is surprising because the search biases of the Adaptive ILP system are initialized by a random process. These biases are normally poor, but the process of natural selection and evolution can successfully evolve a good bias.

It is important to mention that the search bias is rather general because it has reasonable performance on many traditional learning problems using the same bias acquired automatically. This paper illustrates that GP is a plausible technique for implementing a meta-level learning system. For future work, in order to find a general, efficient and effective bias, a large number of learning tasks of different kinds, such as the *member*, *append*, *quick sort*, *ackermann*, *uncle*, and *grandfather* problems, of various characteristics should be used. This adaptive learning approach, though computationally intensive, is rather exciting, as it opens up many opportunities for creating or improving learning algorithms.

Acknowledgments

The first author is sponsored by a scholarship of the Croucher Foundation.

References

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: The University of Michigan Press.

Koza, J. R. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

Lavrac, N., and S. Dzeroski (1994). I*nductive Logic Programming: Techniques and Applications*. London: Ellis Horword.

Michalski, R. S., I. Mozetic, J. Hong, and N. Lavrac. (1986). The multi-purpose incremental learning system AQ15 and its testing application on tree medical domains. In *Proceedings of the National Conference on Artificial Intelligence*, 1041-1045. San Mateo, CA: Morgan Kaufmann.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence* 10: 203-226.

Muggleton, S., and C. Feng. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, 1-14. Tokyo: Ohmsha.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*; 10: 239-266.

Wong, M. L., and K. S. Leung. (1994a). Inductive logic programming using genetic algorithms. In *Advances in Artificial Intelligence - Theory and Application II*, eds. J. W. Brahan and G. E. Lasker, 119-124. Ontario: I.I.A.S.

Wong, M. L., and K. S. Leung. (1994b). Learning first-order relations from noisy databases using genetic algorithms. In *Proceedings of the Second Singapore International Conference on Intelligent Systems.* B159-164.