An Adaptive Knowledge Acquisition System using Generic Genetic Programming

Man Leung Wong

Department of Computing and Decision Sciences

Lingnan University

Tuen Mun

Hong Kong

mlwong@ln.edu.hk

An Adaptive Knowledge Acquisition System using Generic Genetic Programming

Abstract

The knowledge acquisition bottleneck greatly obstructs the development of knowledge-based systems. One popular approach to knowledge acquisition uses inductive concept learning to derive knowledge from examples stored in databases. However, existing learning systems cannot improve themselves automatically. This paper describes an adaptive knowledge acquisition system that can learn first-order logical relations and improve itself automatically. The 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. In this system, 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 the meta-level learner based on generic genetic programming.

It has been demonstrated that the adaptive knowledge acquisition system performs better than FOIL on inducing logical relations from perfect or noisy training examples. The result implies that the search bias evolved by evolutionary learning is better than that of FOIL which is designed by a top researcher in the field. Consequently, Generic genetic programming is a promising technique for implementing a meta-level learning system. The result is very encourage as it suggests that the process of natural selection and evolution can successfully evolve a high performance learning system.

Area: Evolutionary Computation, Genetic Programming, Knowledge Acquisition

1. Introduction

The knowledge acquisition bottleneck greatly obstructs the development of knowledge-based systems. One popular approach to knowledge acquisition uses inductive concept learning to derive knowledge from examples stored in databases. The knowledge acquired can be expressed in different knowledge representations such as first-order logical relations, decision trees, decision lists, and production rules. Existing learning systems such as CART (Breiman et al. 1984), C4.5 (Quinlan

1992), ASSISTANT (Cestnik et al. 1987), AQ15 (Michalski et al 1986), and CN2 (Clark and Niblett 1989) use attribute-value language for representing the training examples and the induced knowledge and allow a finite number of objects in the universe of discourse. This representation limits them to learn only propositional descriptions in which concepts are described in terms of values of a fixed number of attributes.

Dzeroski and Lavrac show that Inductive Logic Programming (ILP) can be used to induce knowledge represented as first-order logical relations (Dzeroski and Lavrac 1993, Dzeroski 1996). ILP is more powerful than traditional inductive learning methods because it uses an expressive first-order logic framework and facilitates the application of background knowledge. In this formalism, domain knowledge represented in the form of relations can be used in the induced relational descriptions of concepts. Moreover, ILP has a strong theoretical foundation from logic programming and computational learning theory.

The task of inducing first-order logical relations can be formulated as a search problem (Mitchell 1982) in a hypotheses space of logical relations. Various approaches (Quinlan 1990; 1996, 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 logical relations from general to specific (or from specific to general) until a consistent relation is found. Each relation in the sequence is obtained by specializing (or generalizing) the previous one. For example, FOIL (Quinlan 1990, 1996) applies a hill climbing search strategy guided by an information-gain heuristic to search relations from general to specific. But these strategies and heuristics are not always applicable because these systems may become trapped in local maxima. In order to overcome this problem, non greedy strategies should be adopted. Moreover, existing ILP systems cannot improve themselves automatically.

In this paper, we describe an adaptive knowledge acquisition system that induces first-order logical relations and improves itself during learning. We formulate

the definitions of inductive concept learning and adaptive knowledge acquisition in the next section. The system is based on a generic genetic programming approach that is presented in Section 3. A generic top-down first-order learning algorithm is described in the next section. The fifth section contains a description of a meta-level learner that induces search bias. The experimentation and some evaluations of the system are reported in Section six. Finally, the conclusion is presented in the last section.

2. INDUCTIVE CONCEPT LEARNING AND ADAPTIVE KNOWLEDGE ACQUISITION

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. The training examples, the hypothesis space, and the background knowledge are represented in first order Horn clause languages (Muggleton 1992). Tradeoffs between expressiveness and efficiency are introduced by some additional restrictions on the three languages representing the examples, the hypothesis space, and the background knowledge. 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 (Lavrac and Dzeroski 1994).

An adaptive knowledge acquisition system is a relational concept learning 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. Since a hypothesis space for learning is defined through the concept description language, the language bias, and the background knowledge, the size and structure of the hypothesis space can be modified by changing the language bias and the background knowledge. The search strategy and heuristics are changed if the system's search biases are modified. Here, we formulate the task of an adaptive knowledge acquisition system in Table 1.

Given:

- A set **E** of positive $\mathbf{E}^+$ and negative $\mathbf{E}^-$ training examples of the target predicate p. Training examples are represented as ground atoms
- 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 **BIASES'**
- A modified set of background knowledge **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 **B'** in **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.
$\mathbf{B} \cup \mathbf{H} \not\models e^-$

Table 1: The definition of adaptive knowledge acquisition

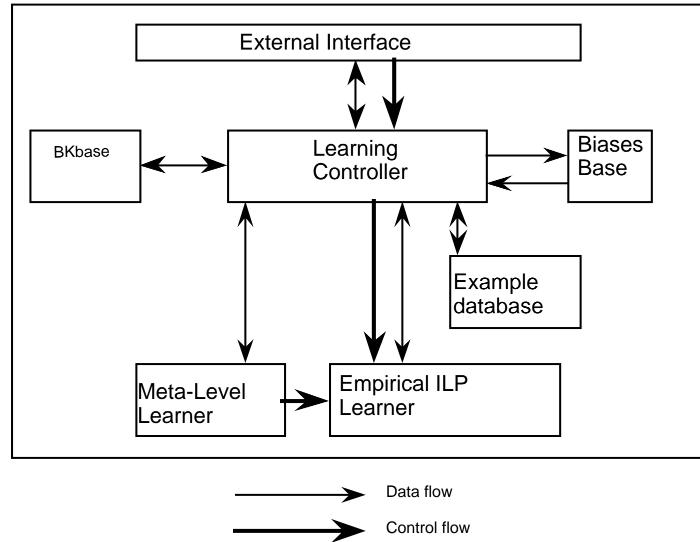

Figure 1: The logical organization of an adaptive knowledge acquisition system

The logical organization of our 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. Biases can be retrieved, added, deleted, and modified through the interface of this knowledge base.

(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*. Since each entity of it is in fact a complex structure representing background knowledge, *BKbase* is implemented using object-oriented techniques.

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

(5) *Empirical ILP learner*: It induces first-order logical relations 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 search bias. In Section 4, a generic top-down first-order learning algorithm is described.

(6) *Meta-level learner*: It learns search biases, language biases, and background knowledge. Search and language biases can be represented declaratively or procedurally. In this paper, we apply GGP to implement the *meta-level learner* that induces procedural biases. The description of GGP is presented in Section 3.

(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. Generic Genetic Programming (GGP)

Generic Genetic Programming (GGP) is a novel approach that combines genetic programming (Koza 1992; 1994, Kinnear 1994) and inductive logic programming (Quinlan 1990; 1996, Muggleton 1992). Using GGP, programs in various programming languages can be evolved. The approach is also powerful enough to handle context-sensitive information and domain-dependent knowledge which can be used to accelerate the learning speed and/or improve the quality of the programs.

GGP can induce programs in various programming languages. This is achieved by accepting or choosing grammars of different languages to produce programs in these languages. Most modern programming languages are specified in the notation of BNF (Backus-Naur form) which is a kind of context-free grammars (CFGs). However, GGP is based on logic grammars because CFGs (Hopcroft and Ullman 1979, Lewis and Rapadimitrion 1981) are not expressive enough to represent context-sensitive information for some languages and domain-dependent knowledge of the target program being induced. This section first introduces the formalism of logic grammars followed by the descriptions of GGP.

3.1. Introduction to logic grammars

Logic grammars are the generalizations of CFGs. Their expressivenesses are much more powerful than those of CFGs, but equally amenable to efficient execution. In this paper, logic grammars are described in a notation similar to that of definite clause grammars (Pereira and Warren 1980, Pereira and Shieber 1987, Sterling and Shapiro 1986). The logic grammar for some simple S-expressions in Table 2 will be used throughout this section.

```
1:    start       ->    [(*], exp(W), exp(W), exp(W), [)].
2:    start       ->    {member(?x,[W, Z])}, [(*], exp-1(?x),
                        exp-1(?x), exp-1(?x), [)].
3:    start       ->    {member(?x,[W, Z])}, [(+], exp-1(?x),
                        exp-1(?x), exp-1(?x), [)].
4:    exp(?x)     ->    [(/ ?x 1.5)].
5:    exp-1(?x)   ->    {random(1,2,?y)}, [(/ ?x ?y)].
6:    exp-1(?x)   ->    {random(3,4,?y)}, [(- ?x ?y)].
7:    exp-1(W)    ->    [(+ (- W 11) 12)].
```

Table 2: A logic grammar

A logic grammar differs from a CFG in that the logic grammar symbols, whether terminal or non-terminal, may include arguments. The arguments can be any term in the grammar. A term is either a logic variable, a function or a constant. A variable is represented by a question mark ? followed by a string of letters and/or digits. A function is a grammar symbol followed by a bracketed n-tuple of terms and

a constant is simply a 0-arity function. Arguments can be used in a logic grammar to enforce context-dependency. Thus, the permissible forms for a constituent may depend on the context in which that constituent occurs in the program. Another application of arguments is to construct tree structures in the course of parsing, such tree structures can provide a representation of the semantics of the program.

The terminal symbols enclosed in square brackets correspond to the set of words of the language specified. For example, the terminal `[(- ?x ?y)]` creates the constituent `(- 1.0 2.0)` of a program if `?x` and `?y` are instantiated respectively to 1.0 and 2.0. Non-terminal symbols are similar to literals in Prolog, `exp-1(?x)` in Table 2 is an example of non-terminal symbols. Commas denote concatenation and each grammar rule ends with a full stop.

The right-hand side of a grammar rule may contain logic goals and grammar symbols. The goals are pure logical predicates for which logical definitions have been given. They specify the conditions that must be satisfied before the rule can be applied. For example, the goal `member(?x, [W, Z])` in Table 2 instantiates the variable `?x` to either `W` or `Z` if `?x` has not been instantiated, otherwise it checks whether the value of `?x` is either `W` or `Z`. If the variable `?y` has not been bound, the goal `random(1, 2, ?y)` instantiates `?y` to a random floating point number between 1 and 2. Otherwise, the goal checks whether the value of `?y` is between 1 and 2.

Domain-dependent knowledge can be represented in logic goals. For example, consider the following grammar rule:

```
a-useful-program    ->   first-component(?X),
                         {is-useful(?X, ?Y)},
                         second-component(?Y).
```

This rule states that a useful program is composed of two components. The first component is generated from the non-terminal `first-component(?X)`. The logic variable `?X` is used to store semantic information about the first component produced. The logic goal then determines whether the first component is useful according to the semantic information stored in `?X`. Domain-dependent knowledge

about which program fragments are useful is represented in the logical definition of this predicate. If the first component is useful, the logic goal `is-useful(?X, ?Y)` is satisfied and some semantic information is stored into the logic variable `?Y`. This information will be used in the non-terminal `second-component(?Y)` to guide the search for a good program fragment as the second component of a useful program.

The special non-terminal `start` corresponds to a program of the language. In Table 2, some grammar symbols are shown in bold-face to identify the constituents that cannot be manipulated by genetic operators. For example, the last terminal symbol **`[)]`** of the second rule is revealed in bold-face because every S-expression must be ended with a ')'. The number before each rule is a label for later discussions. It is not part of the grammar.

3.2. Representations of programs

One of the fundamental contributions of GGP is in the representations of programs in different programming languages appropriately so that initial population can be generated easily and the genetic operators such as reproduction, mutation, and crossover can be performed effectively. A program can be represented as a derivation tree that shows how the program has been derived from the logic grammar. GGP applies deduction to randomly generate programs and their derivation trees in the language declared by the given grammar. These programs form the initial population. For example, the program `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can be generated by GGP given the logic grammar in Table 2. It is derived from the following sequence of derivations:

```
start     =>   [(*] exp(W) exp(W) exp(W) [)]
          =>   [(*] [(/ W 1.5)] exp(W) exp(W) [)]
          =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
               exp(W) [)]
          =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
               [(/ W 1.5)] [)]
          =>   [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

This sequence of derivations can be represented as the derivation tree depicted in Figure 2.

In literature, the terms derivation trees and parse trees are usually used interchangeably. However, we will use the term derivation trees to refer to the tree structures in our framework and the term parse trees to refer to those in GP. The bindings of logic variables are enclosed in a pair of braces. The sub-trees enclosed in a dashed rectangular are frozen. In other words, they are generated by bold-faced grammar symbols and they cannot be modified by genetic operators.

An advantage of logic grammars is that they specify what is a legal program without any explicit reference to the process of program generation and parsing. Furthermore, a logic grammar can be translated into an efficient logic program that can generate and parse the programs in the language declared by the logic grammar (Pereira and Warren 1980, Pereira and Shieber 1987, Abramson and Dahl 1989). In other words, the process of program generation and parsing can be achieved by performing deduction using the translated logic program. Consequently, the program generation and analysis mechanisms of GGP can be implemented using a deduction mechanism based on the logic programs translated from the grammars.

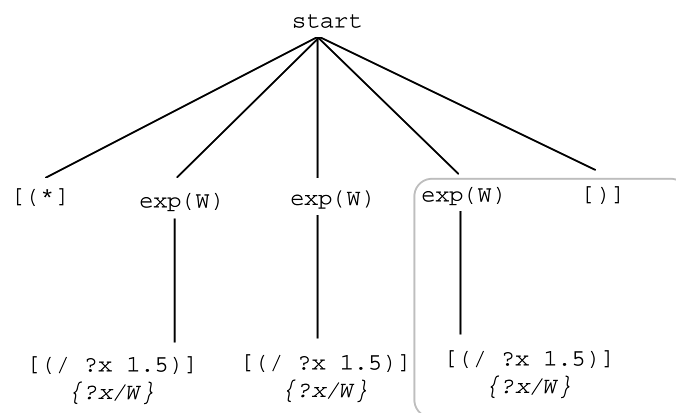

Figure 2: A derivation tree of the S-expression in Lisp
`(* (/ W 1.5) (/ W 1.5) (/ W 1.5))`

This method of translating a logic grammar into a logic program is common in the field of natural language processing (Pereira and Warren 1980, Pereira and Shieber 1987, Abramson and Dahl 1989). The original idea of this approach is to

rephrase the special purpose formalism of CFGs into a general purpose first-order predicate logic (Kowalski 1979, Colmerauer 1978, Pereira and Warren 1980). This approach is further refined and generalized to Definite Clause Grammars (DCGs) which can handle the properties of context-dependency of natural languages effectively. Since DCGs, a kind of logic grammars, can be translated into efficient logic programs automatically, parsers and generators for the corresponding natural languages can be obtained easily. In other words, researchers in the field of natural language processing only declare the grammar for a particular natural language, and the translation process will produce the corresponding parser and generator for them. Moreover, for some cases, the same logic program can be used as both a parser and generator at the same time.

Alternatively, initial programs can be induced by other learning systems such as FOIL (Quinlan 1990; 1996) or given by the user. GGP analyzes each program and creates the corresponding derivation tree.

3.3. The evolution process of GGP

In GGP, populations of programs are genetically bred using the Darwinian principle of survival and reproduction of the fittest along with genetic operations appropriate for creating programs. GGP starts with an initial population of programs generated randomly, induced by other learning systems, or provided by the user. Logic grammars provide declarative descriptions of the valid programs that can appear in the initial population. A fitness function must be defined by the user to evaluate the fitness values of the programs. Typically, each program is run over a set of fitness cases and the fitness function estimates its fitness by performing some statistical operations (e.g. average) to the values returned by this program.

The initial programs in generation 0 are normally incorrect and have poor performances. However, some programs in the population will be fitter than others. Fitness of each program in the generation is estimated and the following process is iterated over many generations until the termination criterion is satisfied. The reproduction, sexual crossover, and asexual mutation are used to create new

generation of programs from the current one. The reproduction involves selecting a program from the current generation and allowing it to survive by copying it into the next generation. Either fitness proportionate or tournament selection can be used.

The crossover is used to create offspring programs from two parental programs selected. Mutation creates a modified offspring program from a parental program selected. Unlike crossover, the offspring program is usually similar to the parent program. Logic grammars are used to constraint the offspring programs that can be produced by these genetic operations.

This algorithm will produce populations of programs which tend to exhibit increasing average of fitness. Finally, GGP returns the best program found in any generation of a run as the result. A high level algorithm of GGP is presented in Table 3.

```
1.    Generate an initial population of programs.
2.    Execute each program in the current population and assign it a
      fitness value according  to the fitness function
3.    If the termination criterion is satisfied, terminate the
      algorithm. The best program found in the run of the algorithm
      is  designated as the result.
4.    Create a new population of programs from the current population
      by applying the reproduction, crossover, and mutation
      operations. These operations are applied to programs selected
      by fitness proportionate or tournament selections.
5.    Rename the new population to the current population.
6.    Proceed to the next generation by branching back to the step 2.
```

Table 3: The high level algorithm of GGP

4. A generic top-down first-order learning algorithm

This section presents a generic top-down first-order learning algorithm based on FOIL (Quinlan 1990, 1996). The algorithm is depicted in Table 4. 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).

```
Input:
   E:              Training examples
   L:              The concept description language
   BIAS_search:    The search bias
   BIAS_lang:      The language bias
   B:              Background knowledge
   T:              The target concept

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

Function LEARNING(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 relation 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 relation P and producing P'. i.e.
    P' := Post-processing(P);
    Return(P');
```

Table 4: A generic top-down first-order learning algorithm

The second step performs the construction of a logical relation. This step employs four local variables: $\mathbf{E}_{current}$ (Current training examples set), $\mathbf{E}'_{current}$ (Updated training examples set), $\mathbf{P}$ (Current relation) and $\mathbf{P}'$ (Modified relation). The main component of this step is the covering loop which implements Michalski's covering algorithm (Michalski et al. 1986a). The covering loop construct a relation 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 relation $\mathbf{P}$ and generate a modified relation $\mathbf{P}'$.

(c) Remove all positive examples from $\mathbf{E}_{current}$ which are covered by $\mathbf{P}'$ with respect to the background knowledge $\mathbf{B}$.

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 relation induced when classifying unseen examples and to simplify the relation.

The covering loop calls the 'Clause-Construct' function which is the core of the generic algorithm. A hill-climbing 'Clause-Construct' algorithm is presented in Table 5. The function constructs a clause $C_n = T \leftarrow l_1, l_2, \ldots, l_n$ 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 $C_i = T \leftarrow l_1, l_2, \ldots, l_i$ is refined by appending a specific literal l_j to its body. A literal l_j is constructed from the background knowledge $\mathbf{B}$ restricted by the concept description language $\mathbf{L}$ 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

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 $\mathbf{B}$. Each training example $<x_1, x_2, ..., x_n>$ is a n-tuple where x_i, $1 \le i \le n$, 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 $X_j = X_k$, $X_j \neq X_k$, $p_m(Y_1, Y_2, ..., Y_{s_m})$ or not $p_m(Y_1, Y_2, ..., Y_{s_m})$.

If the literal contains k new variables, 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 $\mathbf{E}_{current}$ 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 learning algorithm, various learning algorithms can be generated. The performances of a good and a bad learners can be significant different as shown in Sub-section 5.3.

```
Input:
    C:                  An initial clause C = T←
    Ecurrent:           The current training examples
    B:                  Background knowledge
    L:                  The concept description language
    BIASsearch:         The search bias
    BIASlang:           The language bias
Output:
    A clause that covers some positive examples in Ecurrent while excludes all or most
    negatives examples in Ecurrent

Function Clause-Construct(C, Ecurrent, B, L, BIASsearch, BIASlang)

There is a scoring function stored in BIASsearch, 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 Ei of Ecurrent using the clause C w.r.t. B. i.e.
     Ei := Find-Extension(C, Ecurrent, B);
    -Let n_i^+ be the number of positive tuples in Ei;
    -Let n_i^- be the number of negative tuples in Ei;
    -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 BIASlang
        -Set C' = C ∪ {l};
        -Find the extension Ei+1 of Ecurrent using the clause C' i.e.
         Ei+1 := Find-Extension(C', Ecurrent, B);
        -Let n_{i+1}^+ be the number of positive tuples in Ei+1;
        -Let n_{i+1}^- be the number of negative tuples in Ei+1;
        -Let the number of positive tuples in Ei that have been
         represented by one or more tuples in Ei+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, Ecurrent, 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);
```

Table 5: A hill-climbing 'Clause-Construct' algorithm

5. Inducing procedural search biases

In this section, GGP is used in the meta-level learner to induce procedural search biases (i.e. the 'scoring' function). In order to employ GGP, a logic grammar must be defined (Table 6).

In the grammar, the terminal symbols n-pos-i-plus-1, n-neg-i-plus-1, and `n-pos-i` represent respectively n^+_{i+1}, n^-_{i+1} and n^{++}_i. With reference to the algorithms in Tables 4 and 5, assume that $\mathbf{E}_i$ is the extension of current training examples $\mathbf{E}_{current}$ 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 $\mathbf{E}_{i+1}$. The terminal symbol `current-information` is defined as $-\log_2(n^+_i / (n^+_i + n^-_i))$.

```
start          ->    function.
s-exp          ->    term.
s-exp          ->    function.

function       ->    [(], op1, s-exp, [)].
function       ->    [(], op2, s-exp, s-exp, [)].

op1            ->    [ protected-log ].
op2            ->    [ + ].
op2            ->    [ - ].
op2            ->    [ * ].
op2            ->    [ % ].
op2            ->    [ info ].

term           ->    [ n-pos-i-plus-1 ].
term           ->    [ n-neg-i-plus-1 ].
term           ->    [ n-pos-i ].
term           ->    [ current-information ].
term           ->    { random(-10, 10, ?a) },  [ ?a ].
```

Table 6: A logic grammar for learning procedural search bias

The terminal symbols `+`, `-`, and `*` represent functions that perform ordinary addition, subtraction, and multiplication respectively. The symbol `%` represents

function that normally returns the quotient. However, if division by zero is attempted, the function returns 1.0. The symbol `protected-log` is a function that calculates the logarithm of the input argument x if x is larger than zero, otherwise it returns 1.0. The symbol `info` represents the basic function that calculates $-\log_2(X/(X+Y))$ given X and Y as inputs. The logic goal `random(-10, 10, ?a)` generates a random floating point number between -10 and 10 and instantiates `?a` to the random number generated

5.1. The evolution process

The evolution process of the adaptive knowledge acquisition system is depicted in Figure 3. Firstly, the Biases base is initialized with a population of different 'scoring' functions generated randomly using the logic grammar depicted in Table 6. To estimate the fitness of a specific 'scoring' function, it is combined with the generic top-down learner to produce a specific learner. The performance of this specific learner is then evaluated by using a fitness function. This measure is assigned as the fitness of the specific 'scoring' function. GGP employs selection, crossover, and mutation 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

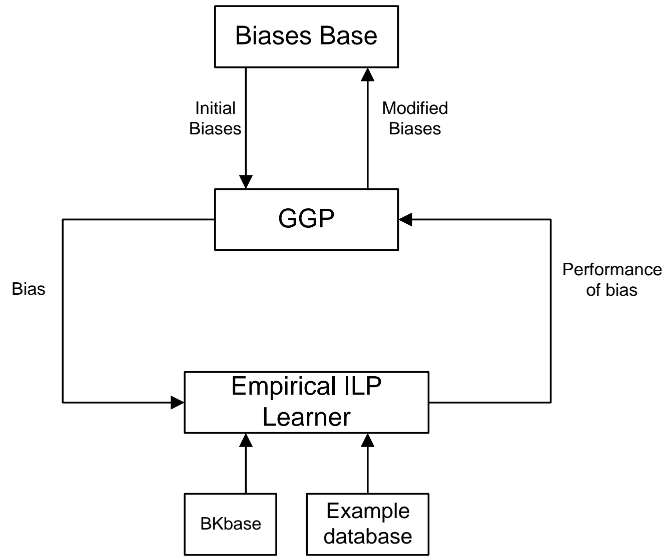

Figure 3: The evolution process of the adaptive knowledge acquisition system

5.2. The experimentation setup

In this paper, learning curves are 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 logical relation 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.

As 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* 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.

5.3. Fitness calculation

Adjusted and normalized fitness values 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 learner to produce a specific learner called Adapted-ILP hereafter. Adapted-ILP induces first-order logical relations using the random example set. The quality of the induced logical relations 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 4.

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 Adapted-ILP induces logical relations from the training set and the relations are evaluated by counting the number of misclassified examples from the testing set. The performance of the Adapted-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 4.

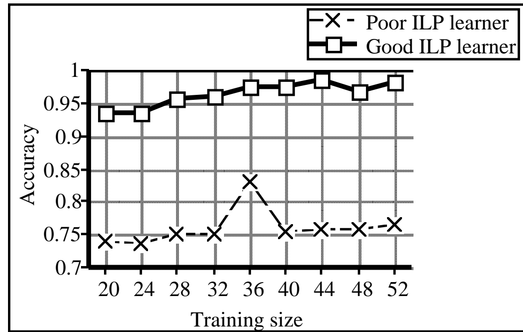

Figure 4: The learning curves of good and poor learner

6. Experimentation and evaluations

This section compares the performance of our system with that of FOIL (Quinlan 1990; 1996). Standard learning tasks in the literature are used in these experiments (Quinlan 1990, Muggleton and Feng 1990).

6.1. The member predicate

The learning curves for this problem are depicted in Figure 5. It is interested to find that our system has higher accuracy than FOIL. The difference is significant at 5% level of significance when the training size is less than 36.

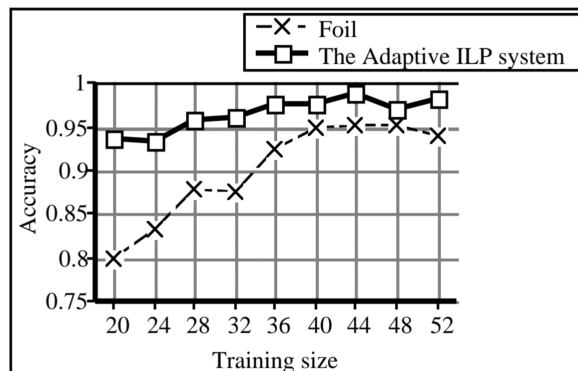

Figure 5: Learning curves for the member problem

6.2. 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 relations 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 6. Our system performs better than FOIL at all noise level.

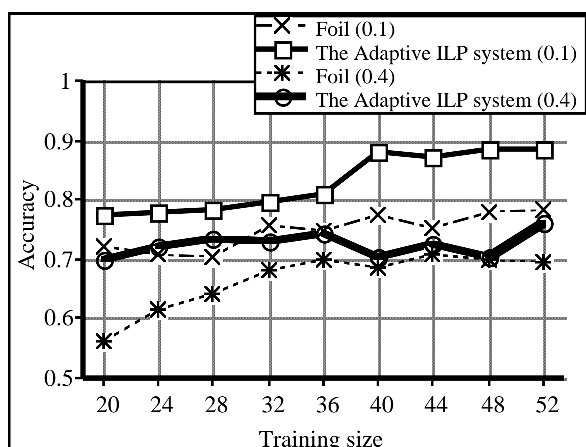

Figure 6: Learning curves for the member problem in a noisy environment

6.3. 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 7. Our system performs better than FOIL when the size of training set is less than 460. The difference is significant at 5% level of significance.

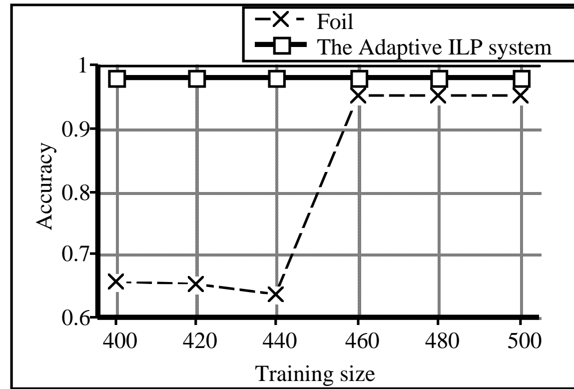

Figure 7: Learning curves for the multiply problem

6.4. 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 8.

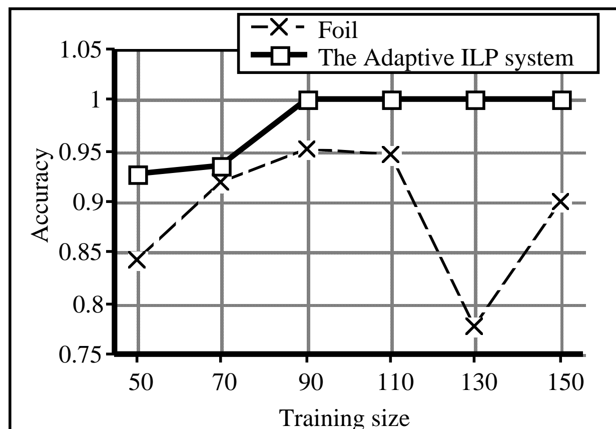

Figure 8: Learning curves for the uncle problem

7. Conclusion

In this paper, we formulate an adaptive knowledge acquisition 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. An implementation of the adaptive knowledge acquisition system has been developed. 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 a meta-level learner implemented by using generic genetic programming.

Generic Genetic Programming (GGP) is a novel, general, and powerful approach of evolving search biases. It is also powerful enough to use logic grammars to represent context-sensitive information and domain-dependent knowledge. The idea of using formal grammars to direct search for knowledge in the hypothesis space or to reduce the size of the space has also been independently studied by other researcher recently (Cohen 1992, Gruau 1996, Whigham 1995; 1996).

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 knowledge acquisition 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 GGP is better than that of FOIL for the learning problems. This result is surprising because the search biases of the adaptive knowledge acquisition 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 GGP is a plausible approach 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, with 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.

Reference

Abramson, H. and Dahl, V. (1989). *Logic Grammars*. Berlin: Springer-Verlag.

Bremen, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984). *Classification and Regression Trees*. Belmont: Wadsworth.

Cestnik, B., Kononenko, J. and Bratko, I. (1987). ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In I. Bratko and N. Lavrac (Ed.), *Progress in Machine Learning*, 31-45. Wilmslow: Sigma Press.

Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*; **3**, 261-283.

Cohen, W. (1992). Compiling Prior Knowledge into an Explicit Bias. In *Proceedings of the Ninth International Workshop on Machine Learning*, 102-110. CA: Morgan Kaufmann.

Colmerauer, A. (1978). Metamorphosis Grammars. In L. Bolc (Ed.), *Natural Language Communication with Computers*. Berlin: Springer-Verlag.

Dzeroski, S. (1996). Inductive Logic Programming and Knowledge Discovery in Databases. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (eds.), *Advances in Knowledge Discovery in Data Mining*, 117-152. Menlo Park, CA: AAAI Press.

Dzeroski, S. and Lavrac, N. (1993). Inductive Learning in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, **5,** 939-949.

Gruau, F. (1996). On Using Syntactic Constraints with Genetic Programming. In P. J. Angeline and K. E. Kinnear, Jr. (Eds.) *Advances in Genetic Programming 2*, 402-417. MA: MIT Press.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. MA: Addison-Wesley.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

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

Kinnear, K. E. Jr., editor (1994). *Advances in Genetic Programming*. Cambridge, MA: MIT Press.

Kowalski, R. A. (1979). *Logic For Problem Solving*. Amsterdam: North-Holland.

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

Lewis, H. R. and Rapadimitrion, C. H. (1981). *Elements of the theory of computation*. NJ: Prentice Hall.

Michalski, R. S., Mozetic, I., Hong, J. and Lavrac, N. (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*, **18**, 203-226.

Muggletion, S., editor (1992). *Inductive Logic Programming*. London: Academic Press.

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

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

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks *Artificial Intelligence*, **13**, 231-278.

Pereira, F. C. N. and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis*. CA: CSLI.

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

Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. Cambridge, MA: MIT Press.

Whigham, P. A. (1996). Search Bias, Language Bias and Genetic Programming. In *Proceedings of the First Genetic Programming Conference*, 230-237. Cambridge, MA: MIT Press.

Whigham, P. A. (1995). Inductive Bias and Genetic Programming. In *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, 461-466. UK: IEE.