An Induction System that Learns Programs in different Programming Languages using Genetic Programming and Logic Grammars

Man Leung Wong
Department of System Engineering and Engineering Management
The Chinese University of Hong Kong
mlwong@se.cuhk.hk

Kwong Sak Leung
Department of Computer Science
The Chinese University of Hong Kong
Hong Kong
ksleung@cs.cuhk.hk

Abstract

Genetic Programming (GP) and Inductive Logic Programming (ILP) have received increasing interest recently. Since their formalisms are so different, these two approaches cannot be integrated easily though they share many common goals and functionalities. A unification will greatly enhance their problem solving power. Moreover, they are restricted in the computer languages in which programs can be induced. In this paper, we have presented a flexible system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that combines GP and ILP. It is based on a formalism of logic grammars. The system can learn programs in various programming languages and represent context-sensitive information and domain-dependent knowledge. The performance of LOGENPRO in inducing logic programs from noisy examples is evaluated. A detailed comparison to FOIL has been conducted. This experiment demonstrates that LOGENPRO is a promising alternative to other inductive logic programming systems and sometimes is superior for handling noisy data. Moreover, a series of examples are used to illustrate that LOGENPRO is so flexible that programs in different programming languages including LISP, Prolog and Fuzzy Prolog. can be induced.

1. Introduction

Program induction generates a computer program that can produce the desired behavior for a given set of situations. Two of the approaches in program induction are Inductive Logic Programming [1] and Genetic Programming [2]. Inductive Logic Programming (ILP) investigates the construction of logic programs from examples and background knowledge while Genetic Programming (GP) is a method of automatically inducing S-expression in LISP to perform specified tasks.

Since their formalisms are so different, these two approaches cannot be integrated easily although their properties and goals are similar. If they can be combined in a common framework, then many of the techniques and theories obtained in one approach can be applied in the other one. The combination can greatly enhance the information exchange between these fields.

Moreover, they are restricted in the computer languages in which programs can be induced. ILP can only learn logic programs. On the other hand, GP represents induced programs as parse trees. In theory, programs in any computer languages such as Prolog, C, and Fortran can be represented as parse trees. Hence, GP should be able to handle them as well. In fact, the process of translating a program in some languages to the corresponding parse tree is not trivial. Since the syntax of LISP is so simple and uniform that the translation process can be achieved easily, programs evolved by GP are usually expressed in LISP.

This paper presents a flexible system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that combines GP and ILP. It can learn programs in various programming languages. The system is also powerful enough to represent context-sensitive information and domain-dependent knowledge. This knowledge can be used to accelerate the learning speed and/or improve the quality of the programs induced.

The next section presents the details of LOGENPRO. Section three demonstrates the application of LOGENPRO in learning S-expressions in LISP. Section four describes a combination of LOGENPRO and FOIL [3] that learns logic programs. The subsequent section further illustrate the flexibility of the framework in inducing programs in Fuzzy Prolog. The last section is the conclusion.

2. The logic grammars based genetic programming system (LOGENPRO)

The LOgic grammars based GENetic PROgramming system (LOGENPRO) can induce programs in various programming languages such as LISP, C and Prolog [4]. Thus, LOGENPRO must be able to accept grammars of different languages and 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 grammar (CFG). However, LOGENPRO is based on logic grammars because CFG is not expressive enough to represent context-sensitive information of the language and domain-dependent knowledge of the target program being induced. Logic grammars are the generalizations of CFG. Their expressivenesses are much more powerful than that of CFG, but equally amenable to efficient execution. In this paper, logic grammars will be

1082-3409/95 $04.00 © 1995 IEEE

described in a notation similar to that of definite clause grammars [5]. The logic grammar in table 1 will be used throughout this section.

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 "meaning" of the program.

The terminal symbols, which are 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 1 is an example of non-terminal symbol. Commas denote concatenation and each grammar rule ends with a full stop.

Table 1. A logic grammar

```
1:start->         [(*], exp(X), exp(X), [)].
2:start->         {member(?x,[X, Y])}, [(*], exp-1(?x),
                  exp-1(?x), [)].
3:start->         {member(?x,[X, Y])}, [(/], exp-1(?x),
                  exp-1(?x), [)].
4:exp(?x)->       [(+ ?x 0)].
5:exp-1(?x)->     {random(0,1,?y)}, [(+ ?x ?y)].
6:exp-1(?x)->     {random(0,1,?y)}, [(- ?x ?y)].
7:exp-1(?x)->     [(+ (- X 11) 12)].
```

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, [X, Y])` in table 1 instantiates the variable ?x to either X or Y if ?x has not been instantiated, otherwise it checks whether the value of ?x is either X or Y. If the variable ?y has not been bound, the goal `random(0, 1, ?y)` instantiates ?y to a random floating point number between 0 and 1. Otherwise, the goal checks whether the value of ?y is between 0 and 1. The special non-terminal `start` corresponds to a program of the language.

The problem of inducing programs can be reformulated as a search for a highly fit program in the space of all possible programs in the language specified by the logic grammar. In LOGENPRO, populations of programs are genetically bred [6] - [9] using the Darwinian principle of survival and reproduction of the fittest along with genetic operations appropriate for processing programs. LOGENPRO 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 high-level algorithm of LOGENPRO is presented in table 2.

Table 2. A high level algorithm of LOGENPRO

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.

The initial programs in generation 0 are normally incorrect and have poor performances. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual crossover 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 one offspring program from the parental programs selected. Logic grammars are used to constraint the offspring programs that can be produced by these genetic operations. Fitness of each program in the new generation is estimated and the above process is iterated over many generations until the termination criterion is satisfied. This algorithm will produce populations of programs which tend to exhibit increasing average of fitness. LOGENPRO returns the best program found in any generation of a run as the result.

One of the contributions of LOGENPRO 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. LOGENPRO 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 `(* (+ X 0) (+ X 0))` can be generated by LOGENPRO given the logic grammar in table 1. Its derivation tree is depicted in figure 1a. Alternatively, initial programs can be induced by other learning systems such as FOIL [3] or given by the user. LOGENPRO analyzes each program and creates the corresponding derivation tree. If the language is ambiguous, multiple derivation trees can be generated. LOGENPRO produces only one tree randomly. For example, the program `(* (+ X 0) (+ X 0))` has multiple derivation trees, two of them are shown in figures 1a and 1b.

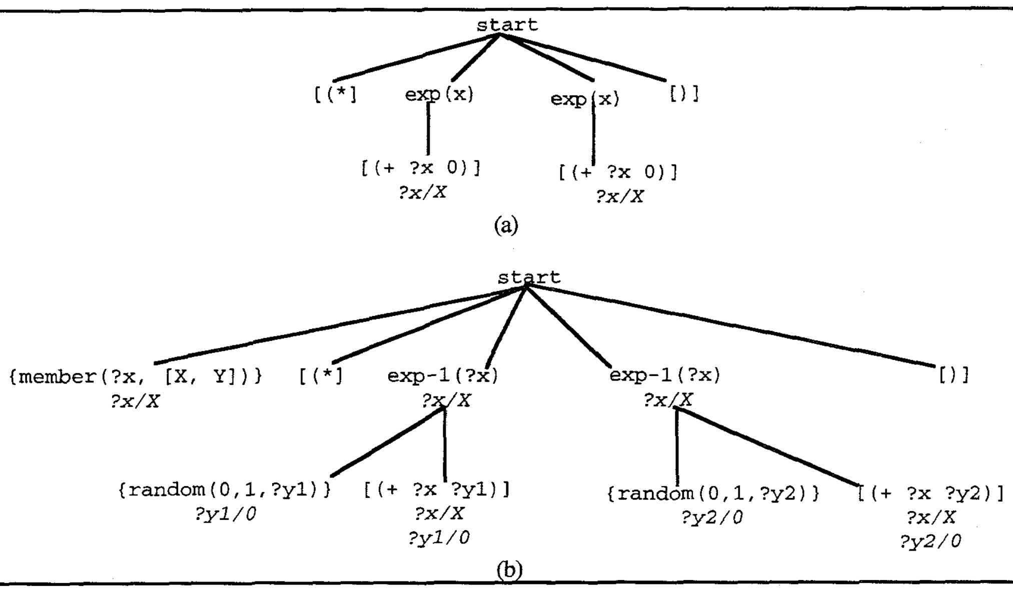

Figure 1. Derivation trees of a program

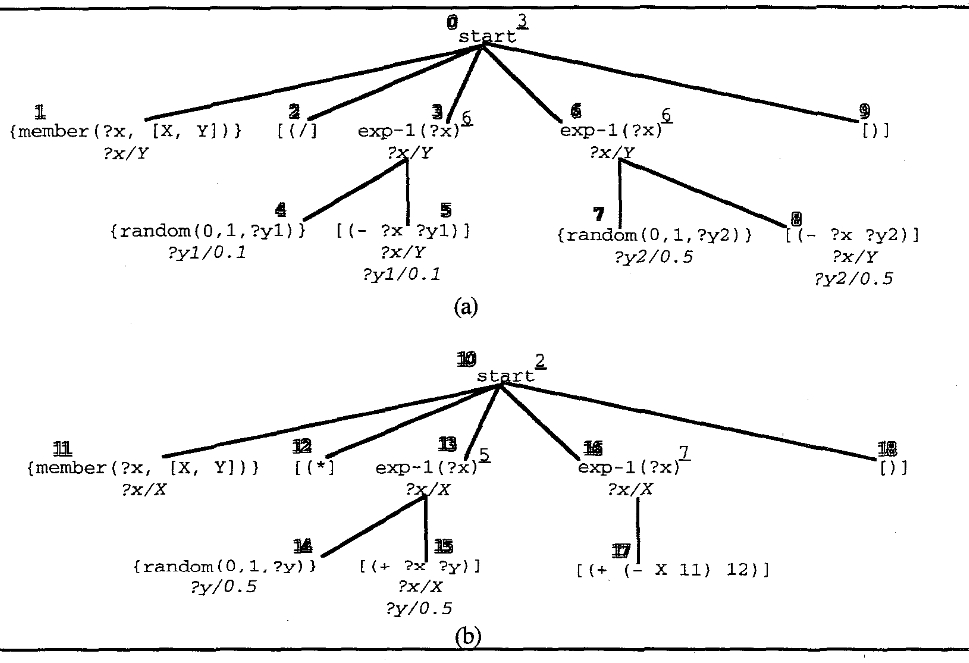

Figure 2. Derivations trees of the parental programs

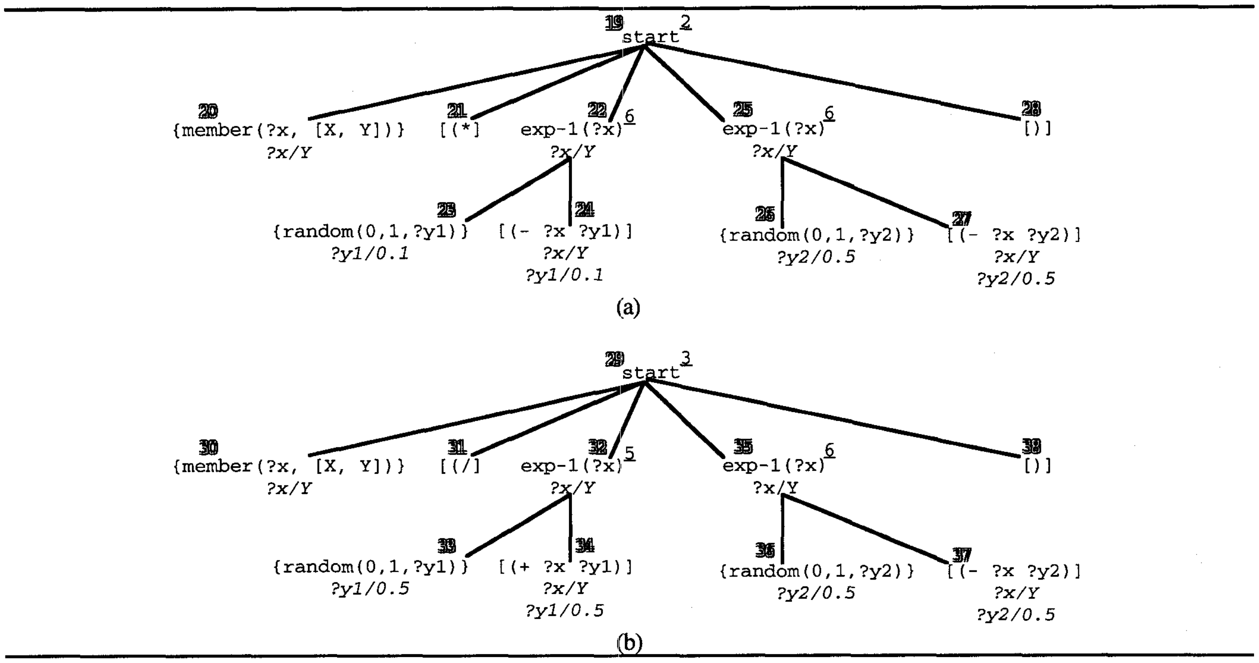
19 start 2
20 {member(?x, [X, Y])} ?x/Y
21 [(*]
22 exp-1(?x) 6 ?x/Y
23 {random(0,1,?y1)} ?y1/0.1
24 [(- ?x ?y1)] ?x/Y ?y1/0.1
25 exp-1(?x) 6 ?x/Y
26 {random(0,1,?y2)} ?y2/0.5
27 [(- ?x ?y2)] ?x/Y ?y2/0.5
28 [)]
(a)
29 start 3
30 {member(?x, [X, Y])} ?x/Y
31 [(/]
32 exp-1(?x) 5 ?x/Y
33 {random(0,1,?y1)} ?y1/0.5
34 [(+ ?x ?y1)] ?x/Y ?y1/0.5
35 exp-1(?x) 6 ?x/Y
36 {random(0,1,?y2)} ?y2/0.5
37 [(- ?x ?y2)] ?x/Y ?y2/0.5
38 [)]
(b)

Figure 3. Derivation trees of offsprings produced by crossover

The crossover is a sexual operation that starts with two parental programs and the corresponding derivation trees. One program is designated as the primary parent and the other one as the secondary parent. The following algorithm is used to produce an offspring program:

1. If there are sub-trees in the primary derivation tree that have not been selected previously, select randomly a sub-tree from these sub-trees using a uniform distribution. The root of the selected sub-tree is called the primary crossover point. Otherwise, terminate the algorithm without generating any offspring.
2. Select another sub-tree in the secondary derivation tree under the constraint that the offspring produced must be valid according to the grammar.
3. If a sub-tree can be found in step 2, complete the crossover algorithm and return the offspring, which is obtained by deleting the selected sub-tree of the primary tree and then impregnating the selected sub-tree from the secondary tree at the primary crossover point. Otherwise, go to step 1.

Consider two parental programs generated by the grammar in table 1, the primary program is (/ (- Y 0.1) (- Y 0.5)) and the secondary program is (* (+ X 0.5) (+ (- X 11) 12)). The corresponding derivation trees are depicted in figures 2a and 2b respectively. In the figures, the shadowed numbers identify sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in parsing the corresponding sub-trees. For example, if the primary and secondary sub-trees are respectively 2 and 12. The valid offspring (* (- Y 0.1) (- Y 0.5)) is obtained and its derivation tree is shown in figure 3a. It is interesting to find that the sub-tree 19 has a label 2. This indicates that the sub-tree is generated by the second grammar rule rather than the third rule applied to the primary parent. The second rule must be used because the terminal symbol [(/] is changed to [(*] and only the second rule can create the terminal [(*].

In another example, the primary and secondary sub-trees are 3 and 13 respectively. The valid offspring (/ (+ Y 0.5) (- Y 0.5)) is produced and the derivation tree is shown in figure 3b. It should be emphasized that the constituent from the secondary parent is changed from (+ X 0.5) to (+ Y 0.5) in the offspring. This must be modified because the logic variable ?x in sub-tree 32 is instantiated to Y in sub-tree 30. This example demonstrates the use of logic grammars to enforce contextual-dependency between different constituents of a program.

LOGENPRO disallows the crossover between the primary sub-tree 6 and the secondary sub-tree 16. The sub-tree 16 requires the variable ?x to be instantiated to X, But, ?x must be instantiated to Y in the context of the primary parent. Since X and Y cannot be unified, these two sub-trees cannot be crossed over.

LOGENPRO has an efficient algorithm to check these conditions before performing any crossover. Thus, only valid offsprings are produced and this operation can be achieved effectively and efficiently.

3. Learning functional program

In this section, we describe how to use LOGENPRO to emulate Koza's GP [2], [10], [11]. Koza's GP has a limitation that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. This limitation leads to the difficulty of inducing even some rather simple and straightforward functional programs. For example, one of these programs calculates the dot product of two given numeric vectors of the same size. Let X and Y be the two input vectors, then the dot product is obtained by the following S-expression:

```
(apply (function +)
    (mapcar (function *) X Y))
```

Let us use this example for illustrative comparison below. To induce a functional program using LOGENPRO, we have to determine the logic grammar, fitness cases, fitness functions and termination criterion. The logic grammar for learning functional programs is given in table 3. In this grammar, we employ the argument of the grammar symbol `s-expr` to designate the data type of the result returned by the S-expression generated from the grammar symbol. For example,

```
(mapcar (function +) X
    (mapcar (function *) X Y))
```

is generated from the grammar symbol `s-expr([list, number, n])` because it returns a numeric vector of size n. Similarly, the symbol `s-expr(number)` can produce `(apply (function *) X)` that returns a number.

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 logic goal `random(-10, 10, ?a)` generates a random floating point number between -10 and 10 and instantiates ?a to the random number generated

Table 3. The logic grammar for the Dot Product problem

```
start                    ->        s-expr(number).
s-expr([list, number, ?n]) -> [ (mapcar (function ],
                                   op2, [ ) ] ,
                                   s-expr([list, number, ?n]),
                                   s-expr([list, number, ?n]),[)].
s-expr([list, number, ?n]) -> [ (mapcar (function ],
                                   op1, [ ) ] ,
                                   s-expr([list, number, ?n]),[)].
s-expr([list, number, ?n]) -> term([list, number, ?n]).
s-expr(number)                -> term(number).
s-expr(number)           ->        [(apply (function ], op2,
                                   [)] ,
                                   s-expr([list, number, ?n]),[)].
s-expr(number)           ->        [ ( ], op2, s-expr(number),
                                   s-expr(number), [ ) ].
s-expr(number)           ->        [ ( ],op1,s-expr(number), [)].
op2                      ->        [ + ].
op2                      ->        [ - ].
op2                      ->        [ * ].
op2                      ->        [ % ].
op1                      ->        [ protected-log ].
term( [list, number, n] ) ->             X.
term( [list, number, n] ) ->             Y.
term( number ) ->                  { random(-10, 10, ?a) }, [?a ].
```

Ten random fitness cases are used for training. Each case is a 3-tuples $\langle X_i, Y_i, Z_i \rangle$, where $1 \leq i \leq 10$, X_i and Y_i are vectors of size 3, and Z_i is the corresponding dot product. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between Z_i and the value returned by the S-expression for X_i and Y_i. A fitness case is said to be covered by an S-expression if the value returned by it is within 0.01 of the desired value. A S-expression that covers all training cases is further evaluated on a testing set containing 1000 random fitness cases. LOGENPRO will stop if the maximum number of generations of 100 is reached or a S-expression that covers all testing fitness cases is found.

For Koza's GP framework, the terminal set T is {X, Y, $\mathbb{R}$} where $\mathbb{R}$ is 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 an individual program in the initial population. The function set F is {protected+, protected-, protected*, protected%, protected-log, vector+, vector-, vector*, vector%, vector-log, apply+, apply-, apply*, apply%}, taking 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 1, 1, 1 and 1 arguments respectively.

The primitive functions protected+, protected- and protected* respectively perform addition, subtraction and multiplication if the two input arguments X and Y are both numbers. Otherwise, they return 0. The function protected% returns the quotient. However, if division by zero is attempted or the two arguments are not numbers, protected% returns 1.0. The function protected-log finds the logarithm of the argument X if X is a number larger than zero. Otherwise, protected-log returns 1.0.

The functions vector+, vector-, vector* and vector% respectively perform vector addition, subtract, multiplication and division if the two input arguments X and Y are numeric vectors with the same size, otherwise they return zero. The primitive function vector-log performs the S-expression:

```
(mapcar (function protected-log) X)
```

if the input argument X is a numeric vector, otherwise it returns zero. The functions apply+, apply-, apply* and apply% respectively perform the following S-expressions if the input argument X is a numeric vector:

```
(apply (function protected+) X),
(apply (function protected-) X),
(apply (function protected*) X) and
(apply (function protected%) X),
```

otherwise they return zero.

The fitness cases, the fitness function and the termination criterion are the same as those used by LOGENPRO. Three experiments are performed. The first one evaluates the performance of LOGENPRO using a population of 100 programs. The other two experiments evaluate the performance of Koza's GP using respectively populations of 100 and 1000 programs. In each experiment, over sixty trials are attempted and the results are summarized in figure 4. From the curves in figure 4, the lower values are better, since LOGENPRO has superior performance than that of GP.

The idea of applying knowledge of data type to accelerate learning has been investigated independently by Montana [12] in his Strongly Typed Genetic Programming (STGP). He presents three examples involving vector and matrix manipulation to illustrate the operation of STGP. However, he has not compare the performance between traditional GP and STGP. Although it is commonly believed that knowledge can accelerate the speed of learning, Pazzani and Kibler [13] shows that inappropriate and/or redundant knowledge can sometimes degrade the performance of a learning system. This section shows that knowledge of data type can be represented in a logic grammar and thus LOGENPRO can emulate the effect of STGP effortlessly.

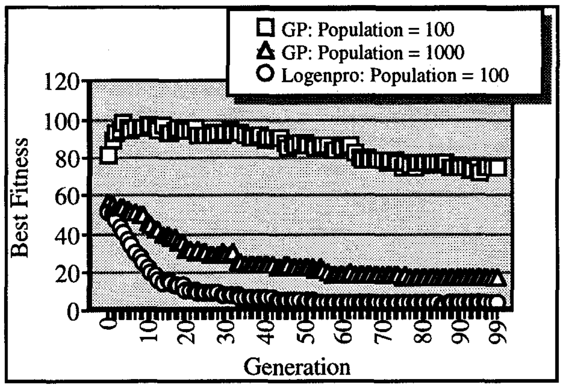

Figure 4. Fitness curves showing best fitness for the Dot Product problem

4. Learning logic program

The task of inducing a logic program can be formulated as a search problem [14] in a hypotheses space of logic programs. Various approaches [1], [3] differ mainly in the search strategy and the heuristics used to guide the search. However, only a few ILP systems such as FOIL [3] and LINUS [15] address the issue of learning from imperfect data which is significant in Knowledge discovery from databases [16]. LOGENPRO is a generalization and extension of GLPS [17], [18]. Genetic Logic Programming System (GLPS) can induce logic programs from noisy examples using Genetic Algorithms [6] - [9]. In this section, we demonstrate that LOGENPRO can emulate GLPS. An empirical comparison of LOGENPRO and FOIL in the domain of the chess endgame problem [3] is presented.

In this domain, the target predicate `illegal(WKf, WKr, WRf, WRr, BKf, BKr)` states whether the position where the white king is at (WKf, WKr), the white rook at (WRf, WRr) and the black king at (BKf, BKr) is not a legal white-to-move position. The background knowledge is represented by two predicates, `adjacent(X, Y)` and `less_than(W, Z)`, indicating that rank/file `X` is adjacent to rank/file `Y` and rank/file `W` is less than rank/file `Z` respectively. LOGENPRO uses the logic grammar in table 4 for this problem. In this grammar, `[adjacent(?X, ?Y)]` and `[less_than(?X, ?Y)]` are terminal symbols. The logic goal `member(?X, [WKf, WKr, WRf, WRr, BKf, BKr ])` will instantiate `?X` to either `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, or `BKr`. The logic variables of the target logic program are `WKf`, `WKr`, `WRf`, `WRr`, `BKf`, and `BKr`.

Table 4. The logic grammar for the chess endgame problem

```
start    ->        clauses.
clauses  ->        clauses,  clauses.
clauses  ->        clause.
clause   ->        consq,  [:-],  antes,  [.].
consq-> [illegal(WKf,  WKr,  WRf,  WRf,  BKf,  BKr)].
antes-> antes,  [,],  antes.
antes-> ante.
ante->  {member(?X,[WKf,  WKr,  WRf,  WRf,  BKf,  BKr])},
        {member(?Y,[WKf,  WKr,  WRf,  WRf,  BKf,  BKr])},
        literal(?X,  ?Y).
literal(?X,  ?Y)   ->        [?X  =  ?Y].
literal(?X,  ?Y)   ->        [ ~ ?X  =  ?Y].
literal(?X,  ?Y)   ->        [ adjacent(?X,  ?Y) ].
literal(?X,  ?Y)   ->        [ ~adjacent(?X,  ?Y) ].
literal(?X,  ?Y)   ->        [ less_than(?X,  ?Y) ].
literal(?X,  ?Y)   ->        [ ~less_than(?X,  ?Y) ].
```

The training set contains 1000 examples (336 positive and 664 negative examples). The testing set has 10000 examples (3240 positive and 6760 negative examples). Different amount of noise is introduced into the training examples in order to study the performances of both systems in learning programs from noisy environment. To introduce n% of noise into argument X of the examples, the value of argument X is replaced by a random value of the same type from a uniform distribution, independent to noise in other arguments. For the class variable, n% positive examples are labeled as negative ones while n% negatives examples are labeled as positive ones. Thus, the probability for an example to be incorrect is $1-\{[(1-n\%)+n\%*\frac{1}{8}]^6*(1-n\%)\}$. In this experiment, the percentages of noise introduced are 5%, 10%, 15%, 20%, 30% and 40%. Thus, the probabilities for an example to be noisy are respectively 27.36%, 48.04%, 63.46%, 74.78%, 88.74% and 95.47%. Background knowledge and testing examples are not corrupted with noise.

A chosen level of noise is first introduced in the training set. Logic programs are then induced from the training set using LOGENPRO and FOIL. Finally, the classification accuracy of the learned programs is estimated on the testing set. For LOGENPRO, the initial population of programs are induced by a variation of FOIL using a portion of the training examples. The population size is 10 and the maximum number of generations for each experiment is 50. Since LOGENPRO is a non-deterministic learning system, the process is repeated for five times on the same training set and the average of the five results is reported as the classification accuracy of LOGENPRO.

Many runs of the above experiments are performed on different training examples. The average results of ten runs are summarized in figure 5. The performances of both systems are compared using paired t-test. From this experiment, the classification accuracy of both systems degrades seriously as the noise level increases. Nevertheless, the classification accuracy of LOGENPRO is better than that of FOIL by at least 5% at the 99.995% confidence interval at all noise levels (except the noise level of 0%). The largest difference reaches 24% at the 20% noise level. One possible explanation of the better performance of LOGENPRO is that the Darwinian principle of survival and reproduction of the fittest is a good noise handling method. It avoids overfitting noisy examples, but at the same time, it can finds interesting and useful patterns from these noisy examples. The experiments demonstrate that LOGENPRO is a promising alternative to other famous inductive logic programming systems and sometimes is superior for handling noisy data.

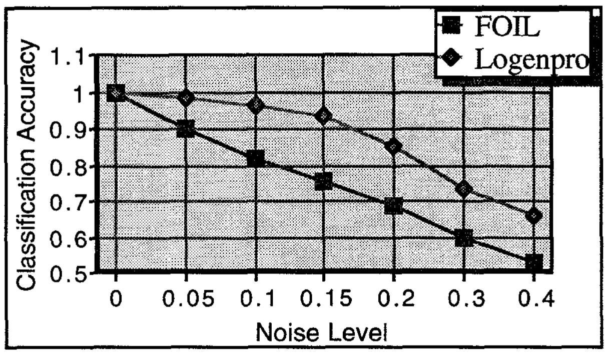

Figure 5. Comparison between LOGENPRO and FOIL

5. Learning programs in Fuzzy Prolog

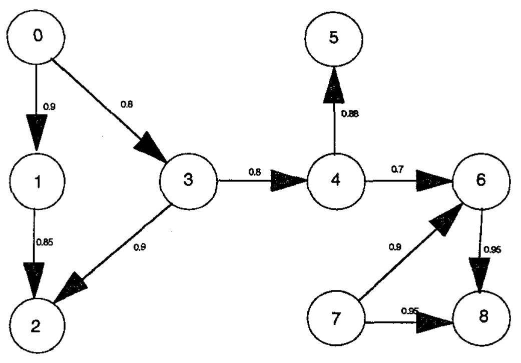

Figure 6. A fuzzy network

The goal of this experiment is to induce a Fuzzy Prolog program that describes the fuzzy relation `can-reach` intensionally. Li and Liu [19] described the detailed definitions of the syntax and semantics of Fuzzy Prolog. LOGENPRO is currently the only system that can learn program in Fuzzy Prolog, to the knowledge of the authors. Consider the fuzzy network in figure 6, this network represents the fuzzy relation `linked-to(X, Y)` that denotes node X is directly linked to node Y with truth value f, where ∈ (0, 1]. In this network, the edges represent the instances of the `linked-to` relation and the number on an edge is the truth value of the corresponding instance. For example, the truth value of the instance `linked-to(0, 1)` is 0.9.

A fuzzy relation is associated with a n-ary fuzzy predicate and can be described extensionally as a set of ordered pairs. Thus the relation `linked-to(X, Y)` can be represented as:

```
linked-to(X, Y) = {(<0,1>, 0.9), (<0,3>, 0.8),
                   (<1,2>, 0.85), (<3,2>, 0.9),
                   (<3,4>, 0.8), (<4,5>, 0.88),
                   (<4,6>, 0.7), (<6,8>, 0.95),
                   (<7,6>, 0.9), (<7,8>, 0.95) }
```

The first element of an ordered pair is a n-tuple of constants that satisfies the associated fuzzy predicate. The second element of the ordered pair is the corresponding truth value. Other fuzzy relations can be obtained from the fuzzy network. One of them is `can-reach(X, Y)` which is represented explicitly as:

```
can-reach(X, Y)={(<0,1>, 0.9), (<0,2>, 0.765),
            (<0,3>, 0.85), (<0,4>, 0.72),
            (<0,5>, 0.648), (<0,6>, 0.567),
            (<0,8>, 0.510), (<1,2>, 0.85),
            (<3,2>, 0.9) (<3,4>, 0.8),
            (<3,5>, 0.72), (<3,6>, 0.63),
            (<3,8>, 0.567), (<4,5>, 0.88),
            ((4,6>, 0.7), ((4,8>, 0.63),
            (<6,8>, 0.95), (<7,6>, 0.9),
            (<7,8>, 0.95) }
```

The negative instances of this relation can be found using the close world assumption [19]. Thus, the set of negative instances is:

```
{(<0,0>, 0), (<0,7>, 0), (<1,0>, 0), (<1,1>, 0),
 (<1,3>, 0), (<1,4>, 0), (<1,5>, 0), (<1,6>, 0),
 (<1,7>, 0), (<1,8>, 0), (<2,0>, 0), (<2,1>, 0),
 (<2,2>, 0), (<2,3>, 0), (<2,4>, 0), (<2,5>, 0),
 (<2,6>, 0), (<2,7>, 0), (<2,8>, 0), (<3,0>, 0),
 (<3,1>, 0), (<3,3>, 0), (<3,7>, 0), (<4,0>, 0),
 (<4,1>, 0), (<4,2>, 0), (<4,3>, 0), (<4,4>, 0),
 (<4,7>, 0), (<5,0>, 0), (<5,1>, 0), (<5,2>, 0),
 (<5,3>, 0), (<5,4>, 0), (<5,5>, 0), (<5,6>, 0),
 (<5,7>, 0), (<5,8>, 0), (<6,0>, 0), (<6,1>, 0),
 (<6,2>, 0), (<6,3>, 0), (<6,4>, 0), (<6,5>, 0),
 (<6,6>, 0), (<6,7>, 0), (<7,0>, 0), (<7,1>, 0),
 (<7,2>, 0), (<7,3>, 0), (<7,4>, 0), (<7,5>, 0),
 (<7,7>, 0), (<8,0>, 0), (<8,1>, 0), (<8,2>, 0),
 (<8,3>, 0), (<8,4>, 0), (<8,5>, 0), (<8,6>, 0),
 (<8,7>, 0), (<8,8>, 0)}
```

Table 5. The logic grammar for the can-reach problem

start	->	clauses.
clauses	->	clauses, clauses.
clauses	->	clause.
clause	->	{random(0, 1, ?A)},
		consq, [<-(?A)], antes, [.].
consq	->	[can-reach(X, Y)].
antes	->	antes, [,], antes.
antes	->	ante.
ante	->	{member(?A,[W, X, Y, Z])},
		{member(?B,[W, X, Y, Z])},
		literal(?A, ?B).
literal(?A, ?B)	->	[linked-to(?A, ?B)].
literal(?A, ?B)	->	[can-reach(?A, ?B)].

The 19 positive and 62 negative instances are used as the fitness cases. The fitness function finds the sum, taken over all 81 fitness cases, of the absolute values of the difference between the desired truth value and the truth value returned by the generated program. A fitness case is said to be covered by a program if the truth value returned is within 0.05 of the desired value. LOGENPRO terminates if the maximum number of generations of 25, is reached or a Fuzzy Prolog program that covers all fitness cases is found. The logic grammar for this problem is shown in table 5. The background knowledge is represented by the fuzzy relation `linked-to(X, Y)` and the predicate `random(0, 1, ?A)` is a logic goal.

A number of trials have been performed using a population size of 100. Correct programs can be found in all trials. The following correct simplified program is found in one trial:

```
can-reach(X, Y)<- (1)
                  linked-to(X, Y).
can-reach(X, Y)<- (0.9)
                  linked-to(X, Z),
                  can-reach(Z, Y).
```

6. Conclusion

We have presented a flexible system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that combines Genetic Programming and Inductive Logic Programming. It is based on a formalism of logic grammars. The system can learn programs in various programming languages and represent context-sensitive information and domain-dependent knowledge. An experiment that employs LOGENPRO to induce a S-expression for calculating dot product has been performed. This experiment illustrates that LOGENPRO, when used with domain knowledge, accelerates the learning of programs. The performance of LOGENPRO in inducing logic programs from imperfect training examples is evaluated using the chess endgame problem. A detailed comparison to FOIL has been conducted. This experiment demonstrate that LOGENPRO is a promising alternative to other inductive logic programming systems and sometimes is superior for handling noisy data. The other experiment illustrates the ability of LOGENPRO in inducing programs Fuzzy Prolog. These experiments prove that LOGENPRO is very flexible.

Acknowledgments

We would like to thank J. R. Quinlan and R. M. Cameron-Jones for providing a public version of FOIL.

Reference

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

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

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

[4] Wong, M. L. and Leung, K. S. (1995). Learning Programs in Different Paradigms using Genetic Programming. In *Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence*. Berlin: Springer-Verlag.

[5] 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**, pp. 231-278.

[6] Holland, J. (1992). *Adaptation in Natural and Artificial Systems*. Cambridge MA: MIT Press.

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

[8] Davis, L. (1987). *Genetic Algorithms and Simulated Annealing*. London: Pitman.

[9] Davis, L. D. (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

[10] Koza, J. R. (1994). *Genetic Programming II*. Cambridge, MA: MIT Press.

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

[12] Montana, D. J. (1993). Strongly Typed Genetic Programming. Bolt, Beranek, and Newman Technical Report no. 7866.

[13] Pazzani, M. and Kibler, D. (1992). The Utility of Knowledge in Inductive learning. *Machine Learning*, **9**, pp. 57-94.

[14] Mitchell, T. M. (1982). Generalization as Search. *Artificial Intelligence*, **18**, pp. 203-226.

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

[16] Piatetsky-Shapiro, G. and Frawley, W. J. (1991). *Knowledge Discovery in Databases*. Menlo Park, CA: AAAI Press.

[17] Wong, M. L. and Leung, K. S. (1994). Inductive Logic Programming Using Genetic Algorithms. In J. W. Brahan and G. E. Lasker (Eds.), *Advances in Artificial Intelligence - Theory and Application II*, pp. 119-124. Ontario: I.I.A.S.

[18] Wong, M. L. and Leung, K. S. (1995). Genetic Logic Programming and Applications. *IEEE Expert*..

[19] Li, D. and Liu, D (1990). A Fuzzy Prolog Database system. Great Britain: Research Studies Press Ltd.