Learning Recursive Functions from Noisy Examples using Generic 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

One of the most important and challenging areas of research in evolutionary algorithms is the investigation of ways to successfully apply evolutionary algorithms to larger and more complicated problems. In this paper, we apply GGP (Generic Genetic Programming) to evolve general recursive functions for the even-n-parity problem from noisy training examples. GGP is very flexible and programs in various programming languages can be acquired. Moreover, it is powerful enough to handle context-sensitive information and domain-dependent knowledge. A number of experiments have been performed to determine the impact of noise in training examples on the speed of learning.

1. Introduction

One of the most important and challenging areas of research in evolutionary algorithms is the investigation of ways to apply them to larger and more complicated problems. One approach to make a large problem more tractable is to discover problem representations automatically. Koza (1994) uses the even-n-parity problem to demonstrate extensively that his approach of hierarchical Automatically Defined Functions (ADFs) can facilitate the solving of the problem.

Koza shows that the even-7-parity problem can be solved using GP with hierarchical ADFs. He finds that about 1440000 functions, I(M, i, z), should be evaluated to obtain at least one solution with 99% probability. In each fitness calculation, 128 fitness cases must be evaluated. Thus, 1440000*128 = 184320000 fitness cases should be processed. Unfortunately, the solutions found can only solve the even-n-parity problem with a particular value of n. If a different value of n is used, GP with hierarchical ADFs must be applied again to find other programs that can solve the new even-n-parity problem.

Clearly, the solution found is not general enough to solve all instances of the even-n-parity problem for all $n \geq 0$. In this paper, we apply a novel Generic Genetic Programming (GGP) approach to evolve general recursive functions for the even-n-parity problem from noisy training examples. The next section describes the GGP approach which is powerful enough to handle context-sensitive information and domain-dependent knowledge. The even-n-parity problem is discussed in Section 3. The subsequent section describes a number of experiments that evaluate the impact of noise in training examples on the rate of inducing general recursive functions. Section 5 is the conclusion.

2. Generic Genetic Programming (GGP)

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 (Dzeroski and Lavrac 1993; Muggletion 1992) and Genetic Programming (Koza 1992, 1994; Kinnear 1994). These approaches are restrictive because Inductive Logic Programming (ILP) can only learn logic programs and Genetic Programming (GP) usually evolves S-expressions in Lisp.

Moreover, 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.

Generic Genetic Programming (GGP) is a novel approach that combines GP and ILP (Wong and Leung

1995b). GGP is a generalization and extension of GLPS (Wong and Leung 1995a). GLPS (the Genetic Logic Programming System) can induce logic programs from noisy examples using Genetic Algorithms (Goldberg 1989; Holland 1975). Using GGP, programs in various programming languages such as Lisp, Prolog, and Fuzzy Prolog can be evolved (Wong and Leung 1995b). 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 (Wong and Leung 1995c).

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 context-free grammar (CFG). However, logic grammars are used in GGP because they are more powerful than that of CFG, but equally amenable to efficient execution. In this paper, the notation of definite clause grammars (DCG) is used (Pereira and Warren 1980). The details of logic grammars and the representation method are described in the Appendixes. On the other hand, the genetic operators are detailed in (Wong and Leung 1996).

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 processing 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 high-level algorithm of GGP is presented in table 1.

Table 1 The high level algorithm of GGP

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.

2.1 Learning logic programs

To illustrate that GGP can learn logic programs, we perform an experiment and compare the performance of GGP and FOIL in inducing logic programs in the domain of the chess endgame problem (Quinlan 1990).

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. GGP uses the logic grammar in table 2 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 2 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 performance 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. 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 GGP and FOIL. Finally, the classification accuracy of the learned programs is estimated on the testing set. For GGP, 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 GGP 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 GGP.

Many runs of the above experiment are performed on different training examples. The average results of ten runs are summarized in figure 1.

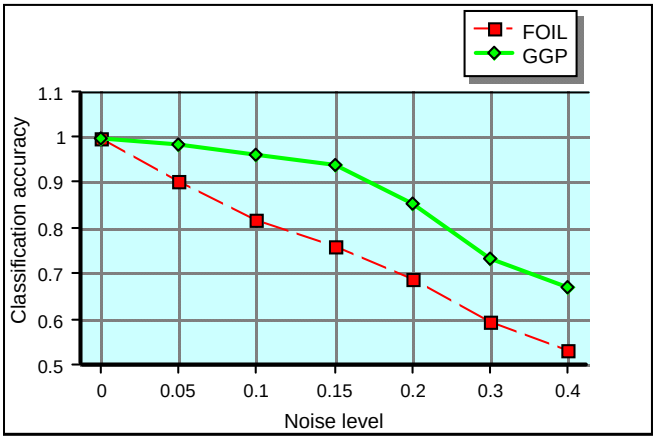

Figure 1Comparison between GGP and FOIL

The performance 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 GGP 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 GGP 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 experiment demonstrate that GGP is a promising alternative to other inductive logic programming systems and sometimes is superior for handling noisy data.

2.2 Learning programs in Fuzzy Prolog

The goal of this experiment is to induce a Fuzzy Prolog program that describes the fuzzy relation `can-reach` intensionally. Li and Liu (Li and Liu 1980) described the detailed definitions of the syntax and semantics of Fuzzy Prolog. Consider the fuzzy network in figure 2, 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 f $\in$ (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.

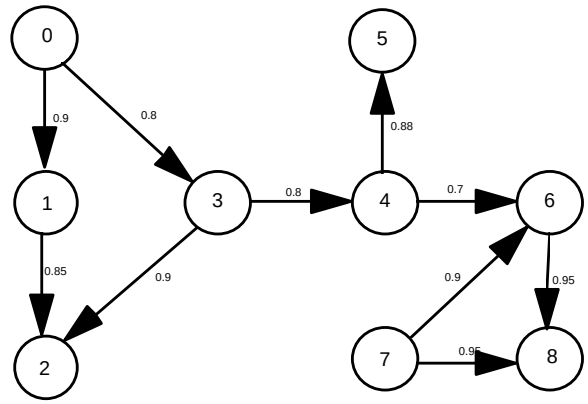

Figure 2A fuzzy network

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 (Li and Liu 1990). 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)}
```

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. GGP 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 3. The background knowledge is represented by the fuzzy relation `linked-to(X, Y)` and the predicate `random(0, 1, ?A)` is a logic goal.

Table 3 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) ].
```

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).
```

3. The Even-n-parity Problem

The boolean even-n-parity function of n boolean input arguments returns true (`T`) if an even number of the arguments are true, otherwise it returns false (`nil`). Koza (1994) uses GP with ADFs to induce hierarchical functions from training examples (fitness cases) to solve the problem. The training set contains all 2^n combinations of the n boolean input arguments. The standardized fitness of an S-expression is the sum of the error between the value returned by the S-expression and the correct value of the even-n-parity function.

Since all 2^n fitness cases, for a particular value of n, are used as the training examples, it is unclear whether GP can discover the regularities of the even-n-parity problem and induce a general function. Moreover, GP can only solve an instance of the even-n-parity problem for a particular value of n. If a different value n is required, GP must be used again to induce another function for the new instance of the problem. A better solution is a recursive function that solves all instances of the problem for all $n \geq 0$.

A general recursive function is given below:

```
(defun parity (L)
 (if (null L)
   T
   (AND
     (OR (first L)
         (parity (rest L)))
     (NAND (first L)
           (parity (rest L))))))
```

To evolve recursive functions using GGP, we have to determine the terminals, the primitive functions, the fitness cases, the fitness function, and the termination criterion. The terminal set is {`L`, `T`, `nil`} where `L` is the input argument of the recursive function to be learned, `T` and `nil` are boolean truth values. The argument `L` is a list of boolean values and any number of boolean values can exist in the list `L`. The set of primitive functions is {`AND`, `OR`, `NAND`, `NOR`, `ifnil`, `first`, `rest`, `parity`}

The boolean functions `AND`, `OR`, `NAND`, and `NOR` take two boolean input arguments and return one boolean value. The function `ifnil` takes three arguments. The first argument must be an S-expression that returns a list of boolean values. The last two arguments must be S-expressions that produce boolean output values. The function `ifnil` checks whether the first input argument returns an empty list. If the list is empty, `ifnil` returns the boolean value of the second S-expression as the output value of the function, otherwise it returns the boolean value of the third S-expression.

The primitive function `first` takes a list of boolean values as its argument and returns the first boolean value of the list if the list is not empty. Otherwise, the function generates an exception signal to indicate that an illegal operation has been attempted to get the first element from an

empty list. The primitive function `rest` must take a list of n boolean values, for any value of n $\geq$ 0, as its argument. If the input list is not empty, it returns a list containing the last n-1 elements of the input list, otherwise, the function generates an exception signal to indicate an illegal action has been tried. The primitive function `parity` takes a list of boolean values as its input and returns a boolean value. This function recursively calls the recursive function being evolved by GGP.

There are three data types: BOOLEAN, LIST, and SMALLER-LIST, that can be used to specify the primitive functions. The data type LIST contains lists of n boolean values, for any value of n $\geq$ 0. The data type SMALLER-LIST contains lists of m boolean values for any value of m < n. The logic grammar for this problem is given in table 4.

In this grammar, we employ the argument of the non-terminal grammar symbol `s-expr` to designate the data type of the result returned by the S-expression generated from the grammar symbol. For example, the S-expression (AND (OR T nil) (NAND nil T)) returning a boolean value can be generated from the non-terminal grammar symbol `s-expr(BOOLEAN)`. Similarly, the non-terminal grammar symbol s-expr(LIST) can produce the S-expression (rest (rest L)) that returns a list of boolean values.

The terminal grammar symbols **`[T]`**, **`[nil]`**, and **`[L]`** in rules 12, 13, and 17 of the grammar form the terminal set of the problem. The terminal grammar symbols **`[AND]`**, **`[OR]`**, **`[NAND]`**, and **`[NOR]`** in rules 20, 21, 22, and 23 represent primitive functions that perform ordinary boolean operations. Rule 14 of the grammar specifies that these primitive functions take two boolean input arguments and return one boolean value.

The terminal symbol **`[first]`** in rule 16 represents the primitive function that returns the first element of a list of boolean values. Rule 16 also specifies that the primitive function takes a list of boolean values as its argument and returns a boolean value. The terminal symbol **`[rest]`** in rule 19 represents the primitive function that takes a list of n boolean values as its argument and returns a list containing the last n - 1 elements of the input list. Rule 19 also declares the arity of the function and the data types of its argument and output value.

Since an item of the SMALLER-LIST data type is also a list of boolean values, it should belong to the LIST data type. This fact is specified in rule 18. This example shows that a hierarchy of data types can be declared easily using grammar rules.

The primitive function represented by the terminal symbol **`[parity]`** in rule 15 must take a list of the SMALLER-LIST data type. This rule avoids a non-terminating recursive function such as: (defun parity (L) (ifnil L T (parity L))) to be evolved by GGP.

Table 4 The logic grammar for the even-n-parity problem

```
11:start                  ->
      [ (defun parity (L) ],
      [ (ifnil L T ],
      s-expr(boolean), [ )) ].
12:s-expr(BOOLEAN)        ->    [ T ].
13:s-expr(BOOLEAN)        ->    [ nil ].
14:s-expr(BOOLEAN)        ->
      [ ( ],op, s-expr(BOOLEAN),
      s-expr(BOOLEAN), [ ) ].
15:s-expr(BOOLEAN)        ->
      [ ( ], [ parity ],
      s-expr(SMALLER-LIST), [ ) ].
16:s-expr(BOOLEAN)        ->
      [ ( ], [ first ], s-expr(LIST),
      [ ) ].
17:s-expr(LIST)           ->    [ L ].
18:s-expr(LIST)           ->
      s-expr(SMALLER-LIST).
19:s-expr(SMALLER-LIST)   ->
      [ ( ],[ rest ],
      s-expr(LIST),[ ) ].
20:op                     ->    [ AND ].
21:op                     ->    [ OR ].
22:op                     ->    [ NAND ].
23:op                     ->    [ NOR ].
```

The idea of applying knowledge of data type to accelerate learning has been investigated independently by Montana (1994) in the Strongly Typed Genetic Programming (STGP). The previous paragraphs show that knowledge of data types can be represented easily using logic grammars and thus GGP can emulate the effect of STGP easily as a type of knowledge among many others.

The first rule of the grammar represents the domain-specific knowledge that the recursive function to be evolved must return `T` if the input argument `L` is an empty list, and the outermost statement of the function must be the correct base statement. The results of the experiments are discussed in the next section.

4. Experiments

Three experiments have been conducted. They differ in the fitness cases used. In each experiment, the population size is 500 and the maximum number of generations is 50. The probabilities of performing crossover and mutation are respectively 0.7 and 0.1. The maximum depth of derivation trees generated in the initial population is 12. The maximum depth of trees produced by crossover and mutation is also 12. All experiments are repeated for 60 times.

The first experiment evaluates the ability of GGP in inducing recursive functions for the even-n-parity problem. The even-0-, 2-, and 3- parity problems are used in the training process. The training set contains all 13 fitness cases from these even-parity problems. The standardized

fitness value of an evolved function is the total number of misclassifications on the 13 fitness cases. The evolution terminates if the maximum number of generations of 50 is reached or a function that classifies all fitness cases correctly is found. In order to avoid the problem caused by inefficient functions, an execution time limit is enforced. After executing 100 primitive functions, if the evolved function fails to find a result for a fitness case, it will be terminated. In this case, it is assumed that the function will misclassify the corresponding fitness case.

It is possible that an evolved function will generate exceptions during its execution for some fitness cases, because it is illegal to perform the first/rest operation on an empty list. If the function produces an exception, it is assumed that it will misclassify the corresponding fitness cases.

In the 60 trials, GGP successfully evolves 16 functions that classify all fitness cases correctly. The generated functions are then tested on the even-i-parity problems, where i $\in$ {0, 1, 2, 4, 5, 6, 7, 8, 9, 10}. All of them can successfully solve all the problems. They are further analyzed manually and it is found that these 16 functions are correct recursive functions for the general even-n-parity problem.

The I(M, i, z) for z of 99% reaches a minimum value of 335000 at generation 9 (Koza 1992). Since there are 13 fitness cases in the training set, 335000*13 = 4355000 fitness cases should be processed. Thus, GGP can find a general, recursive function for the even-n-parity problem very efficiently. On the other hand, GP with hierarchical ADFs evaluates 184320000 fitness cases to find a function that solves the even-7-parity problem only. In other words, GGP can solve the even-7-parity problem about 42 times faster than GP with hierarchical ADFs.

The second experiment evaluates the ability of GGP in inducing recursive functions for the even-n-parity problem from noisy training examples. The even-0-, 2-, and 3- parity problems are used in the training process. The training set contains all 13 fitness cases from these even-parity problems. To introduce noise into the training examples, one of them is randomly selected and the result of the selected example is modified from T to nil or from nil to T. The fitness function and the termination criterion are the same as those of the first experiment.

GGP successfully evolves 12 correct recursive functions for the general even-n-parity problem. The I(M, i, z) for z of 99% reaches a minimum value of 450000 at generation 9. Since there are 13 fitness cases in the training set, 450000*13 = 5850000 fitness cases should be processed. It is found that 1.34 times effort must be used if a noisy training set is employed compared to the first experiment. On the other hand, GGP can solve the even-7-parity problem about 32 times faster than GP with hierarchical ADFs.

The third experiment is similar to the second experiment. The only difference between them is that 2 of the 13 training examples are changed to introduce more noise into the training examples. The selected examples are modified from T to nil or from nil to T.

GGP successfully evolves 5 correct recursive functions for the general even-n-parity problem. The I(M, i, z) for z of 99% reaches a minimum value of 765000 at generation 16. Since there are 13 fitness cases in the training set, 765000*13 = 9945000 fitness cases should be processed. By comparing with the result of the second experiment, it is found that 1.7 times effort must be used if there is more noise in the training set. On the other hand, GGP can solve the even-7-parity problem about 18.53 times faster than GP with hierarchical ADFs.

5. Conclusion

In this paper, we described a method of evolving recursive functions for the even-n-parity problem from noisy training examples. Three experiments have been performed to study the impact of noise in training examples on the speed of learning recursive functions. The numbers of fitness cases processed to induce general recursive functions with 99% probability are summarized in table 5. These experiments demonstrated that GGP can evolve recursive functions from noisy training examples. However, more computation effort is required if more noise exists in training examples.

Table 5 The numbers of fitness cases processed to induce general recursive functions with 99% probability

The first experiment (0% noise)	4355000
The second experiment (7.7% noise)	5850000
The third experiment (15.4% noise)	9945000

For future work, we will use the techniques of the inductive learning systems such as THESYS (Summers 1977) and ADATE (Olsson 1995) so that GGP can induce recursive functional programs more efficiently.

Acknowledgments

The research is sponsored by the RGC Earmarked research grant of UGC reference number CUHK 486/95E.

Appendix A: Logic Grammars

Logic grammars are the generalizations of CFGs. Their expressiveness is much more powerful than those of CFGs, but equally amenable to efficient execution (Pereira and Warren 1980). The logic grammar for some simple S-expressions is given in table 6.

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.

Table 6 A logic grammar

```
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)].
```

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 6 is an example of non-terminal symbol. 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, if the variable `?y` has not been bound, the goal `random(1, 2, ?y)` in table 6 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 6, 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 end with a '`)`', and thus it is not necessary to modify the '`)`' symbol. The number before each rule is a label for later discussions and is not part of the grammar.

Appendix B: 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 6. 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 3.

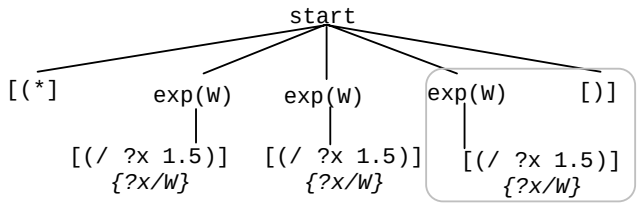

Figure 3A derivation tree of the S-expression in Lisp `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))`

The bindings of logic variables are shown in italic font and enclosed in a pair of braces. The sub-trees enclosed in a rectangular are frozen. In other words, they are generated by

bold-faced grammar symbols and they cannot be modified by genetic operators.

One 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). In other words, the process of program generation and parsing can be achieved by performing deduction using the produced 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. In the following paragraphs, we discuss the method of implementing GGP using a Prolog-like logic programming language.

The differences between the logic programming language used and Prolog are listed as follows:

- A variable is represented by a question mark ? followed by a string of letters and/or digits.
- The elements of a list can be separated by either commas or spaces. For example, `[a b c]` and `[a, b, c]` are equivalent.
- A pair of `'|'` is used to represent a frozen terminal symbol. For example, the symbol **`[)]`** in the second rule of the grammar in table 6 is translated into `|)|`.
- A pair of braces encloses a sequence of logic goals appearing in a logic grammar.
- If there are a number of clauses C_1, C_2, ..., C_n that match a goal G, the ordering of evaluating these clauses is determined randomly.

Using the difference list approach (Sterling and Shapiro 1986), a grammar rule of the form:

A_0 -> A_1, A_2, ..., A_n.

is translated into a logic program clause of the form:

A_0' :- A_1', A_2', ..., A_n'.

in the logic programming language. Here, if A_i , for some i between 0 and n, is a non-terminal with M arguments, then A_i' is a literal with M+3 arguments. The predicate symbols of A_i and A_i' are the same. For example, A_i is translated into `exp(?X, ?Tree, ?S`$_j$`, ?S`$_{j+1}$`)`, for some j, if A_i is `exp(?X)`. The literal `exp(?X, ?Tree, ?S`$_j$`, ?S`$_{j+1}$`)` states that the sequence of symbols between `?S`$_j$ and `?S`$_{j+1}$ is a sentence of the category represented by the non-terminal symbol `exp(?X)`. The derivation tree of the sentence is stored in the logic variable `?Tree`.

A terminal symbol such as `[a b c]` is translated to a literal with 3 arguments: `connect([a b c], ?S`$_j$`, ?S`$_{j+1}$`)`, for some j. The predicate `connect` is defined as:

```
connect(?A, ?S0,?S1):-
    append(?A, ?S1, ?S0).
```

This predicate declares that the list of symbols stored in the logic variable `?A` can be found in the sequence of symbols between `?S0` and `?S1`.

If A_k, for some k between 1 and n, is a pair of braces enclosing a sequence of pure logic goals, i.e., A_k has the form of {G_0, G_1, ..., G_m}, then A_k' is obtained from A_k by removing the pair of braces.

For example, the grammar depicted in table 6 can be translated into the logic program presented in table 7. In the clause 1' of the logic program shown in table 7, the compound term `tree(start, [(*], ?E1, ?E2, frozen(?E3), |)|)` indicates that it is a tree with a root labeled as `start`. The children of the root include the terminal symbol `[(*]`, a tree created from the non-terminal `exp(W)`, another tree created from the non-terminal `exp(W)`, a frozen tree generated from the non-terminal **`exp(W)`**, and the frozen terminal `|)|`.

Thus, a derivation tree can be generated randomly by issuing the following query:

```
?- start(?T, ?S, []).
```

This goal can be satisfied by deducing a sentence that is in the language specified by the grammar. One solution is:

```
?S = [(* (/ W 1.5) (/ W 1.5)
      (/ W 1.5))]
```

and the corresponding derivation tree is:

```
?T = tree(start, [(*],
          tree(exp(W), [(/ W 1.5)]),
          tree(exp(W), [(/ W 1.5)]),
          frozen(tree(exp(W),
          [(/ W 1.5)])), |)|)
```

This is exactly a representation of the derivation tree shown in figure 3. In fact, the bindings of all logic variables and other information are also maintained in the derivation trees to facilitate the genetic operations that will be performed on the derivation trees.

Alternatively, initial programs can be induced by other learning systems such as FOIL (Quinlan 1990) or given by the user. GGP analyzes each program and creates the corresponding derivation tree. If the language is ambiguous, multiple derivation trees can be generated. GGP produces only one tree randomly.

Using the logic program in table 7, a given program such as `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can be analyzed using the following query:

```
?-start(?T,
[(* (/ W 1.5) (/ W 1.5) (/ W 1.5))],
[]).
```

The given program is correct if the above goal can be satisfied and the corresponding derivation tree will be bound to the logic variable `?T`. Since the logic grammar in table 6 is ambiguous, the corresponding logic program may produce multiple derivation trees for a given program. Since the search strategy of the underlying deduction mechanism selects randomly one clause to explore with backtracking from all unifiable clauses, the sequence of generating the derivation trees of a particular program is also random. Consequently, GGP takes the first tree returned from the query to represent the given program.

Table 7 A logic program obtained from translating the logic grammar presented in table 6

```
1':start(tree(start,[(*],?E1,?E2,
         frozen(?E3), |)|), ?S0, ?S5)
      :- connect([(*], ?S0, ?S1),
         exp(W, ?E1, ?S1, ?S2),
         exp(W, ?E2, ?S2, ?S3),
         exp(W, ?E3, ?S3, ?S4),
         connect([)], ?S4, ?S5).
2':start(tree(start,
         {member(?x, [W, Z])},
         [(*],  ?E1,  ?E2,  frozen(?E3),
         |)|),?S0, ?S5)
      :- member(?x, [W, Z]),
         connect([(*], ?S0, ?S1),
         exp-1(?x, ?E1, ?S1, ?S2),
         exp-1(?x, ?E2, ?S2, ?S3),
         exp-1(?x, ?E3, ?S3, ?S4),
         connect([)], ?S4, ?S5).
3':start(tree(start,
         {member(?x, [W, Z])},
         [(+], ?E1, ?E2, frozen(?E3),
         |)|),?S0, ?S5)
      :- member(?x, [W, Z]),
         connect([(+], ?S0, ?S1),
         exp-1(?x, ?E1, ?S1, ?S2),
         exp-1(?x, ?E2, ?S2, ?S3),
         exp-1(?x, ?E3, ?S3, ?S4),
         connect([)], ?S4, ?S5).
4':exp(?x, tree(exp(?x),
       [(/ ?x 1.5)]),?S0, ?S1)
      :- connect([(/ ?x 1.5)],?S0,?S1).
5':exp-1(?x, tree(exp-1(?x),
         {random(1,2,?y)},
         [(/ ?x ?y)]),?S0, ?S1)
      :- random(1, 2, ?y),
         connect([(/ ?x ?y)], ?S0, ?S1).
6':exp-1(?x, tree(exp-1(?x),
         {random(3,4,?y)},
         [(- ?x ?y)]),?S0, ?S1)
      :- random(3, 4, ?y),
         connect([(- ?x ?y)], ?S0, ?S1).
7':exp-1(W,
         tree(exp-1(W),
         [(+ (- W 11) 12)]),?S0, ?S1)
      :- connect([(+ (- W 11) 12)],
         ?S0, ?S1).
```

Bibliography

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

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

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

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

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

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

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

Montana, D. J. (1994) Strongly Typed Genetic Programming. Technical Report 7866, Bolt, Beranek, and Newman.

Muggleton, S. (1992) Inductive Logic Programming. In S. Muggletion (ed.), *Inductive Logic Programming*, pp. 3-27. London: Academic Press.

Olsson, R. (1995) Inductive Functional Programming using Incremental Program Transformation. *Artificial Intelligence*, **74**, pp. 55-81.

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.

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

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

Summers, P. D. (1977) A Methodology for LISP Program Construction from Examples. *JACM*, **24**, pp. 161-175.

Wong, M. L. and Leung, K. S.(1995a) Inducing Logic Programs with Genetic Algorithms: The Genetic Logic Programming System, *IEEE Expert*, **9**, no. 5. pp. 68-76.

Wong, M. L. and Leung, K. S. (1995b). An Induction System that Learns Programs in different Programming Languages using Genetic Programming and Logic Grammars. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*. pp. 380-387. CA: IEEE Computer Society Press.

Wong, M. L. and Leung, K. S. (1995c). Applying Logic Grammars to Induce sub-functions in Genetic Programming. In *Proceedings of the 1995 IEEE International Conference on Evolutionary Computing.*

Wong, M. L. and Leung, K. S. (1996). Evolving recursive functions for the even-parity problem using genetic programming. In P. J. Angeline and K. E. Kinnear, Jr. (Eds.) *Advances in Genetic Programming 2*. Cambridge MA: MIT Press.