Applying logic grammars to induce sub-functions in 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

Genetic Programming (GP) is a method of automatically inducing S-expression in LISP to perform specified tasks. The problem of inducing programs can be reformulated as a search for a highly fit program in the space of all possible programs. This paper presents a framework in which the search space can be specified declaratively by a user. Its application in inducing sub-functions is detailed. The framework is based on a formalism of logic grammars and it is implemented as a system called LOGENPRO (the LOgic grammar based GENetic PROgramming system). The formalism is 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 system is also very flexible and programs in various programming languages can be acquired.

Automatic discovery of sub-functions is one of the most important research areas in Genetic Programming. An experiment is used to demonstrate that LOGENPRO can emulate Koza's Automatically Defined Functions (ADF). Moreover, LOGENPRO can employ knowledge such as argument types in a unified framework. The experiment shows that LOGENPRO has superior performance to that of Koza's ADF when more domain-dependent knowledge is available.

1. Introduction

Genetic Programming (GP) is a method of automatically inducing S-expression in LISP to perform specified tasks [3] [4]. The problem of inducing programs can be reformulated as a search for a highly fit program in the space of all possible programs [5]. This space is determined by the syntax of S-expression in LISP and the sets of terminals and functions. Thus, the search space is fixed once the terminals and functions are decided.

This paper presents a framework in which the search space can be specified declaratively by a user and describes its application in inducing sub-functions. For most complex problems, the problem representation has enormous influence on the difficulty of solving the problem using GP. Thus appropriate primitives and terminals must be determined to represent the problem. It is important and challenging that appropriate problem representation can be induced automatically because higher level primitives (sub-functions) can transform a hard problem into an easy one.

This framework is based on a formalism of logic grammars and it is implemented as a system called LOGENPRO (the LOgic grammar based GENetic PROgramming system). The formalism is 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 formalism is also very flexible and programs in various programming languages can be acquired.

We present the formalism of logic grammars and LOGENPRO in the next section. In section three, we demonstrate the application of various knowledge to accelerate the learning of sub-functions in the framework. Section four is the conclusion.

2. The LOgic grammars based GENetic PROgramming system (LOGENPRO)

LOGENPRO can induce programs in various programming languages. This is achieved by accepting or choosing grammars of different languages in order 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 LOGENPRO because they are much more powerful than that of CFG, but equally amenable to efficient execution. In this paper, the notation of definite clause grammars (DCG) is used [7]. The details of logic grammars are described in the Appendix.

In LOGENPRO, populations of programs are genetically bred [1] [2] 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 1.

Table 1: The 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.

3. Learning sub-functions using LOGENPRO

Automatic discovery of problem representation primitives is certainly one of the most challenging research areas in Genetic Programming. Automatically Defined Functions (ADF) is one of the approaches that have been proposed to acquire problem representation primitives automatically [3] [4]. In the ADF approach, each program in the population contains an expression, called the result producing branch, and definitions of one or more sub-functions which may be invoked by the result producing branch. The result producing branch is evaluated to produce the fitness of the program. A constrained syntactic structure and some special genetic operators are required for the evolution of the programs. To employ the ADF approach, the user must provide explicit knowledge about the number of available automatically defined sub-functions, the number of arguments of each sub-functions, and the allowable terminal and function sets for each sub-function.

In this section, we demonstrate how to use LOGENPRO to emulate Koza's ADF approach. Koza's ADF 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. In this experiment, LOGENPRO is expected to learn a sub-function that calculates dot product and employ this sub-function in the main program. In other words, it is expected to induce the following S-expression:

```
(progn
 (defun ADF0 (arg0 arg1)
  (apply (function +)
   (mapcar (function *) arg0 arg1)))
 (+ (ADF0 X Y) (ADF0 Y Z)))
```

Table 2: Logic grammar for the sub-function problem

```
 start     ->        [(progn (defun ADF0 (arg0 arg1)],
                     s-expr2(number), [])],
                     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])
          ->         term([list, number, ?n]).
s-expr(number)
          ->         [ (apply (function ], op2, [ ) ] ,
                     s-expr([list, number, ?n]),[ ) ].
s-expr(number)
          ->         [ ( ], op2, s-expr(number),
                     s-expr(number), [ ) ].
s-expr(number)
          ->         [ (ADF0 ],
                     s-expr([list, number, ?n]),
                     s-expr([list, number, ?n]), [ ) ].
term([list, number, n])       ->        X.
term([list, number, n])       ->        Y.
term([list, number, n])       ->        Z.
s-expr2([list, number, ?n])
          ->         [ (mapcar (function ], op2, [ ) ] ,
                     s-expr2([list, number, ?n]),
                     s-expr2([list, number, ?n]),[ ) ].
s-expr2([list, number, ?n])
          ->         term2([list, number, ?n]).
s-expr2(number)
          ->         [ (apply (function ], op2, [ ) ] ,
                     s-expr2([list, number, ?n]),[ ) ].
s-expr2(number)
          ->         [ ( ], op2, s-expr2(number),
                     s-expr2(number), [ ) ].
term2([list, number, n])      ->        arg0.
term2([list, number, n])      ->        arg1.
op2       ->         [ + ].
op2       ->         [ - ].
op2       ->         [ * ].
```

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 2. 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.

Ten random fitness cases are used for training. Each case is a 4-tuples $\langle \mathbf{X}_i, \mathbf{Y}_i, \mathbf{Z}_i, R_i \rangle$, where $1 \leq i \leq 10$, $\mathbf{X}_i$, $\mathbf{Y}_i$ and $\mathbf{Z}_i$ are vectors of size 3, and R_i is the corresponding desired result. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between R_i and the value returned by the S-expression for $\mathbf{X}_i$, $\mathbf{Y}_i$ and $\mathbf{Z}_i$. 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 is reached or a S-expression that covers all testing fitness cases is found.

For Koza's ADF framework, the terminal set T_0 for the automatically defined function (ADF0) is {arg0, arg1} and the function set F_0 is {protected+, protected-, protected*, vector+, vector-, vector*, apply+, apply-, apply*}, taking 2, 2, 2, 2, 2, 2, 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 functions vector+, vector- and vector* respectively perform vector addition, subtract and multiplication if the two input arguments X and Y are numeric vectors with the same size, otherwise they return zero. The functions 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) and
 (apply (function protected*) X),
```

otherwise they return zero.

The terminal set T_r for the result producing branch is {X, Y, Z} and the function set F_r is {protected+, protected-, protected*, vector+, vector-, vector*, apply+, apply-, apply*, ADF0}, taking 2, 2, 2, 2, 2, 2, 1, 1, 1 and 2 arguments respectively. The fitness cases, the fitness function and the termination criterion are the same as the ones used by LOGENPRO. We evaluate the performance of LOGENPRO and Koza's ADF using populations of 100 and 1000 programs respectively.

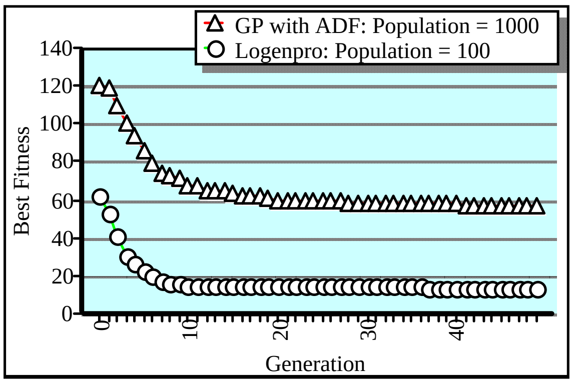
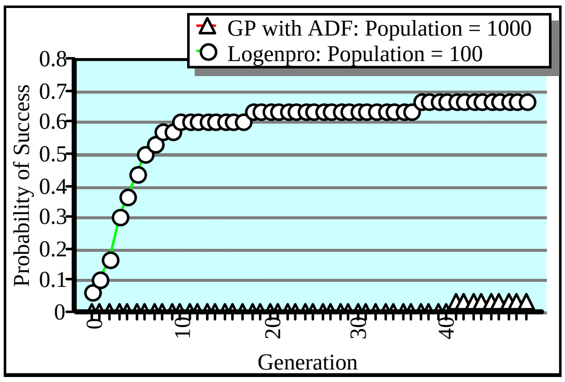
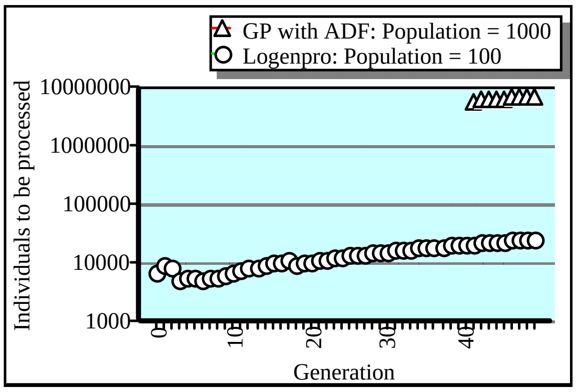


Fig. 1. Fitness curves showing best fitness for the sub-function problem

Thirty trials are attempted and the results are summarized in figures 1 and 2 Figure 1 shows, by generation, the fitness (error) of the best program in a population. These curves are found by averaging the results obtained in thirty different runs using various random number seeds and fitness cases. From these curves, LOGENPRO has superior performance to that of Koza's ADF. The curves in figure 2a show the experimentally observed cumulative probability of success, P(M, i), of solving the problem by generation i using a population of M programs. The curves in figure 2b show the number of programs I(M, i, z) that must be processed to produce a solution by generation i with a probability z of 0.99. The curve for LOGENPRO reaches a minimum value of 4900 at generation 6. On the other hand, the minimum value of I(M, i, z) for ADF is 5712000 at generation 41. This experiment clearly shows the advantage of LOGENPRO. By employing various knowledge about the problem being solved, LOGENPRO can find a solution much faster than ADF and the computation (i.e. I(M, i, z)) required by LOGENPRO is much smaller than that of ADF.



(a)



(b)

Fig. 2. Performance curves showing (a) cumulative probability of success P(M, i) and (b) I(M, i, z) for the sub-function problem

The idea of applying knowledge of data type to accelerate learning has been investigated independently by Montana [6] 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 compared the performance between traditional GP and STGP. Moreover, his STGP cannot be used with ADF nor to specify any domain specific knowledge. One advantage of LOGENPRO is that it can emulate the effects of STGP and ADF simultaneously and effortlessly.

4. Conclusion

Genetic Programming induces programs by searching a highly fit program in the space of all possible programs. We have proposed a framework that the search space can be declared explicitly. This framework is based on a formalism of logic grammars. To implement the framework, a system called LOGENPRO (the LOgic grammar based GENetic PROgramming system) has been developed. The formalism can represent context-sensitive information and domain-dependent knowledge.

Automatic discovery of sub-functions is one of the most important research areas in Genetic Programming. In Koza's ADF, the user must provide explicit knowledge about the number of available sub-functions, the number of arguments of each sub-functions, and the allowable terminal and function sets for each sub-function. An experiment has been performed to demonstrate that LOGENPRO can emulate Koza's ADF and represent the knowledge easily. Moreover, LOGENPRO can employ other knowledge such as argument types in a unified framework. This experiment shows that LOGENPRO has superior performance to that of Koza's ADF when more domain-dependent knowledge is available.

5. Reference

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

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

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

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

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

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

[7] 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. Appendix

A logic grammar (table 3) 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 logical variables, 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 (meaning) of the program.

Table 3: A simple 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 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 3 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, [X, Y])` in figure 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 number before each rule is a label for later discussions. It is not part of the grammar.