Learning Programs in Different Paradigms using Genetic Programming

Man Leung Wong
Department of Computing and Decision Sciences
Lingnan University, Tuen Mun
Hong Kong
Fax: 852-28922442
mlwong@ln.edu.hk

Kwong Sak Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong
Fax: 852-26035024
ksleung@cse.cuhk.edu.hk

Abstract

Genetic Programming (GP) is a method of automatically inducing programs by representing them as parse trees. In theory, programs in any computer languages can be translated to parse trees. Hence, GP should be able to handle them as well. In practice, 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 framework that programs in various programming languages can be acquired. 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. 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 knowledge of data types, accelerates the learning of programs. Other experiments have been done to illustrate the ability of LOGENPRO in inducing programs in difference programming languages including Prolog and C. These experiments prove that LOGENPRO is very flexible.

1 Introduction

Genetic Programming (GP) is a method of automatically inducing programs by representing them 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 framework that programs in various programming languages can be acquired. This framework is based on a formalism of logic grammars and a system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) is developed. The formalism 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.

We present the formalism of logic grammars and the details of LOGENPRO in section two. Section three demonstrates the application of LOGENPRO in learning S-expressions in Lisp. The subsequent two sections further illustrate the flexibility of the framework in inducing programs in Prolog and C. Section six is the conclusion.

2 The Logic Grammars Based Genetic Programming System (LOGENPRO)

LOGENPRO can induce programs in various programming languages such as Lisp, Prolog, and C. Thus, it 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 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 (Pereira and Warren 1980).

A logic grammar differs from a CFG in that the logic grammar symbols, whether terminal or non-terminal, may include arguments (Table 1). The arguments can be any term in the grammar. A term is either a logical 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 or to construct representation "meaning" in the course of parsing.

```
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)].
```

Table 1. A logic grammar

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 figure 1 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, 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. 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.

In LOGENPRO, populations of programs are genetically bred (Goldberg 1989) 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.

---

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

Table 2. A high level algorithm of Logenpro

3 Learning Functional Programs

In this section, we describe how to use LOGENPRO to emulate Koza's GP (Koza 1992; 1994). 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

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.

```
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].
```

Table 3. A logic grammar for the Dot Product problem

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.

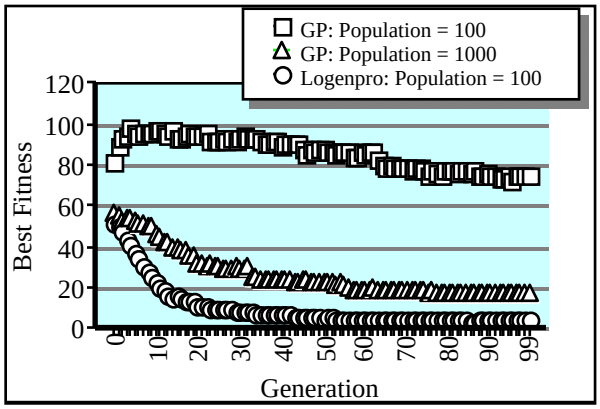
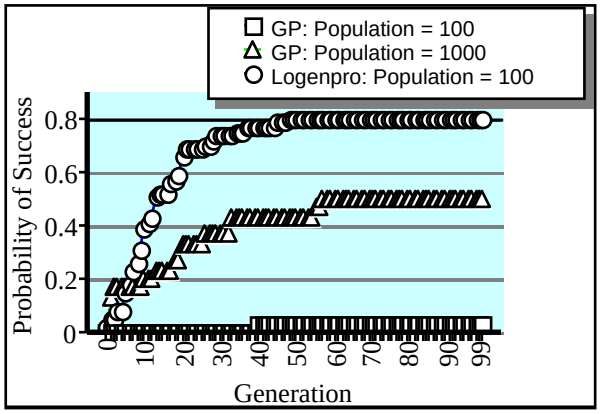
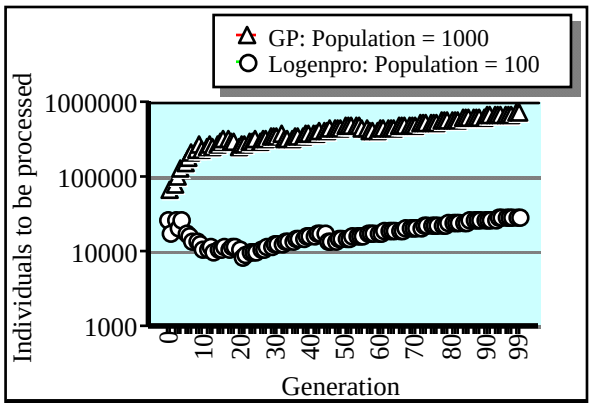


Figure 1. Fitness curves showing best fitness for the Dot Product problem

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 figures 1 and 2. From the curves in figures 1, LOGENPRO has superior performance than that of GP.



(a)



(b)

Figure 2. Performance curves showing (a) cumulative probability of success P(M, i) and (b) I(M, i, z) for the Dot Product problem

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 (Koza 1992; 1994). Throughout this paper, the probability z is set to 0.99. The curve for GP with a population of 100 programs is not depicted because the values is extremely large. For the LOGENPRO curve, I(M, i, z) reaches a minimum value of 8800 at generation 21. On the other hand, the minimum value of I(M, i, z) for GP with population size of 1000 is 66000 at generation 1. LOGENPRO can find a solution much faster than GP and the computation (i.e. I(M, i, z)) required by LOGENPRO is much smaller than that of GP.

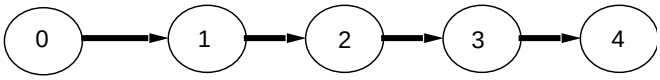
The idea of applying knowledge of data type to accelerate learning has been investigated independently by Montana (1993) in his Strongly Typed Genetic Programming (STGP). He presents three examples involving vector and matrix manipulation to illustrate the operation of STGP. However, he has not compare the performance between traditional GP and STGP. Although it is commonly believed that knowledge can accelerate the speed of learning, Pazzani and Kibler (1992) shows that inappropriate and/or redundant knowledge can sometimes degrade the performance of a learning system. One advantage of LOGENPRO is that it can emulate the effect of STGP effortlessly.

4 Learning Logic Programs

In this section, we describe how to use LOGENPRO to induce logic programs. To induce a target logic program, we have to determine the logic grammar, the fitness function and other major parameters such as the population size, the maximum number of generations, and the probabilities of applying various genetic operations.

For concept learning (DeJong et al. 1993, Janikow 1993), each individual logic program in the population can be evaluated in terms of how well it covers positive examples and excludes negative examples. Thus, the fitness functions for concept learning problems calculate this measurement. Typically, each logic program is run over a number of training examples so that its fitness is measured as the total number of misclassified positive and negative examples. Sometimes, if the distribution of positive and negative examples is extremely uneven, this method of estimating fitness is not good enough to focus the search. For example, assume that there are 2 positive and 10000 negative examples, if the number of misclassified examples is used as the fitness value, a logic program that deduces everything are negative will have very good fitness. Thus, in this case, the fitness function should find a weighted sum of the total numbers of misclassified positive and negative examples.

The modified Quinlan's network reachability problem is used as a demonstration. The problem is originally proposed by Quinlan (Quinlan 1990), the domain involves a directional network such as the one depicted as follows:



The structural information of the network is the literal `linked-to(A, B)` denoting that node `A` is directly linked to node `B`. The extension of `linked-to(A, B)` is {`<0, 1>, <1, 2>, <2, 3>, <3, 4>`}. Here, the learning task is to induce a logic program that determines whether a node `A` can reach another node `B`. This problem can also be formulated as finding the intensional definition of the relation `can-reach(A, B)` given its extension. Its extensional definition is {`<0, 1>, <0, 2>, <0, 3>, <0, 4>, <1, 2>, <1, 3>, <1, 4>, <2, 3>, <2, 4>, <3, 4>`}. The tuples of this relation are the positive training examples and {`<0, 0>, <1, 0>, <1, 1>, <2, 0>, <2, 1>, <2, 2>, <3, 0>, <3, 1>, <3, 2>, <3, 3>, <4, 0>, <4, 1>, <4, 2>, <4, 3>, <4, 4>`} are the negative examples.

```
start                 ->     clauses.
clauses               ->     clauses, clauses.
clauses               ->     clause.
clause                ->     consq, [:-], antes, [.].
consq                 ->     [can-reach(A, B)].
antes                 ->     antes, [,], antes.
antes                 ->     ante.
ante                  ->     {member(?x,[A, B, C])},
                             {member(?y,[A, B, C])}
                             {not-equal(?x, ?y)},
                             literal(?x, ?y).
literal(?x, ?y)       ->     [linked-to(?x, ?y)].
literal(?x, ?y)       ->     [can-reach(?x, ?y)].
```

Table 4. The logic grammar for the modified Quinlan's network reachability problem

The logic grammar for this experiment is shown in table 4. In this experiment, the population size is 1000. The standardized fitness is the total number of misclassified training examples. The maximum number of generations is 50. Five runs are performed and LOGENPRO can find a perfect program that covers all positive examples while excludes all negative ones within a few generations. One program found is:

```
can-reach(A, B) :- linked-to(C, B),linked-to(A, C).
can-reach(A, B) :- linked-to(A, B),linked-to(A, C).
can-reach(A, B) :- can-reach(A, C),can-reach(C, B).
```

This program can be simplified to

```
can-reach(A, B) :- linked-to(A, C),linked-to(C, B).
can-reach(A, B) :- linked-to(A, B).
can-reach(A, B) :- can-reach(A, C),can-reach(C, B).
```

The first clause of this program declares that a node `A` can reach node `B` if there is another node `C` that directly connects them. The second clause declares that a node `A` can reach a node `B` if they are directly connected. The third clause is recursive, it expresses that a node `A` can reach a node `B` if there is another node `C`, such that `C` is reachable from `A` and `B` is reachable from `C`. In fact, this program is semantically equivalent to the standard solution

```
can-reach(A, B) :- linked-to(A, B).
can-reach(A, B) :- linked-to(A, C),can-reach(C, B).
```

This experiment demonstrates that LOGENPRO can learn recursive program naturally and effectively. Recursive functions are difficult to learn in Koza's GP (Koza 1992; 1994), this experiment shows the advantage of LOGENPRO over GP. Figure 3 depicts the best, average and worst standardized fitnesses for increasing generations. These fitnesses curves are obtained by averaging the corresponding fitness values obtained from five different runs.

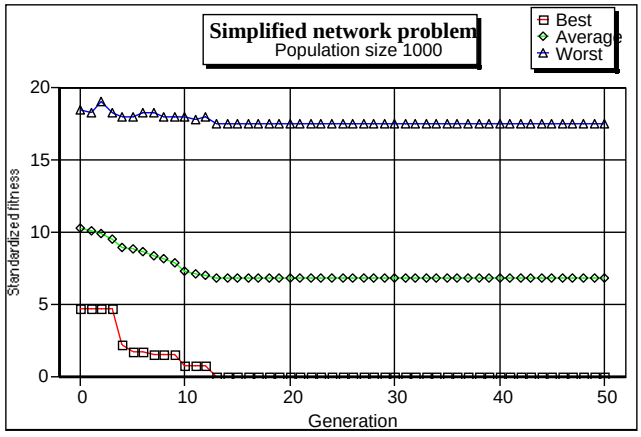


Figure 3. Fitness curves for the modified network reachability problem

5 Learning Programs in C

The target program calculates the function value f(X, Y) for the two input arguments and outputs the result. The function f(X, Y) is $((X+Y)^2-Y)$ and the population size used in this experiment is 500. The ten fitness cases are 3-tuples $<X_i, Y_i, f(X_i, Y_i)>$, where $1 \leq i \leq 10$ and X_i, Y_i are random integer between 0 and 10. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between $f(X_i, Y_i)$ and the value returned by the generated C program using X_i and Y_i as the inputs. A fitness case is said to be covered by a program if the value returned by it is within 0.01 of the desired value. LOGENPRO terminates if the

maximum number of generations, which is 50, is reached or a C program that covers all fitness cases is found.

The logic grammar for this problem is shown in table 5. In this grammar, only simple assignment statement can be generated. This restriction is enforced only to limit the size of the search space for the problem so that solutions can be found using the available computational resources. In fact, the search space will be extremely large if the complete grammar for the C programming language is used. In this grammar, the symbol `preamble` produces statements that declare and initialize variables used in the program. On the other hand, the symbol `outputs` creates statement that prints the final result of the program.

```
start               ->    preamble, statements, outputs.
statements          ->    statements, statements.
statements          ->    statement.
statement           ->    id, [=], expression, [;].
expression          ->    [(], expression, op, expression, [)].
expression          ->    id.
op                  ->    [+].
op                  ->    [-].
op                  ->    [*].
id                  ->    [X].
id                  ->    [Y].
id                  ->    [Z].
preamble            ->    [#include <stdio.h>],
                          [#include <stdlib.h>],
                          [main(argc, argv)],
                          [int argc; char **argv;],
                          [{ int X Y; float Z;],
                          [   X = atoi(argv[1]);],
                          [   Y = atoi(argv[2]);],
                          [   Z = 0.0;].
outputs             ->    [   printf("\n%f", Z)].
```

Table 5. The logic grammar for learning programs in C

In one successful run of LOGENPRO, the following correct C program is found in generation 4:

```
#include <stdio.h>
#include <stdlib.h>
main(argc, argv)
int argc; char **argv;
{ int X, Y; float Z;
  X = atoi(argv[1]);
  Y = atoi(argv[2]);
  Z = 0.0;
  Z = (((X-Z)*X)+((Y*Y)+(((X+X)*Y)-Y)));
  printf("\n%f", Z);}
```

The program is correct because the assignment statement $Z = (((X - Z) * X) + ((Y * Y) + (((X + X) * Y) - Y)))$ can be simplified to

$Z = X^2 + Y^2 + 2XY - Y$ as the variable Z is initialized to 0.0. The statement can be further simplified to $Z = (X + Y)^2 - Y$ which is the desire statement. It should be emphasized that the goal of this section is to demonstrate the possibility of learning programs in some imperative languages. The symbolic regression problem is deliberately constructed as simple as possible so as to illustrate the point clearly.

Twenty trials are attempted using different random number seeds and fitness cases. The results are summarized in figures 4 and 5. Figure 4 shows, by generation, the fitness of the best program in a population. Figure 5 shows the performance curves when the population size M is 500 and the probability z is 0.99. The value of I(M, i, z) reaches a minimum value of 21000 at generation 5.

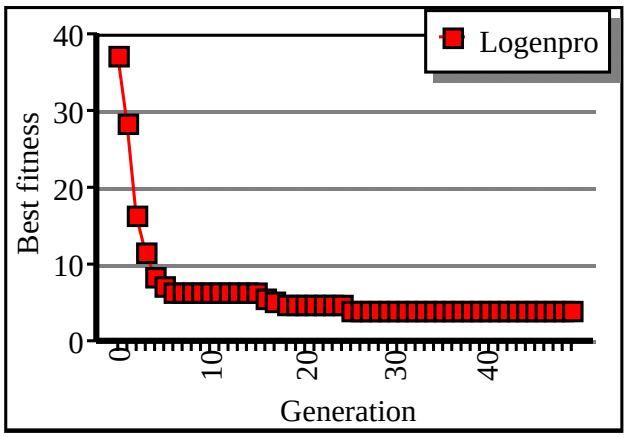


Figure 4. Fitness curve for the problem of inducing a C program

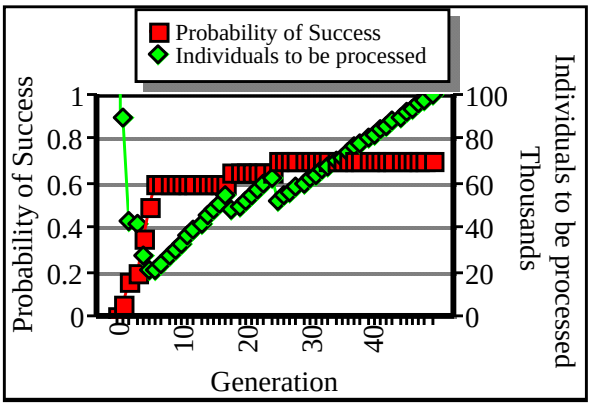


Figure 5. Performance curves for the problem of inducing programs in C

6 Conclusion

We have proposed a flexible framework that programs in various programming languages can be acquired. 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. 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 knowledge of data types, accelerates the learning of programs. Other experiments have been done to illustrate the ability of LOGENPRO in inducing programs in difference programming languages including Prolog and C. These experiments prove that LOGENPRO is very flexible.

Reference

DeJong, K. A., Spears, W. M. and Gordon, D. F. (1993). Using Genetic Algorithms for Concept Learning, *Machine Learning*, **13**, 161-188.

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

Janikow, C. Z. (1993). A Knowledge-Intensive Genetic Algorithm for Supervised Learning, *Machine Learning*, **13**, 189-228.

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

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

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

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

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.