Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming

Man Leung Wong

Department of Computing and Decision Sciences

Lingnan University

Tuen Mun

Hong Kong

mlwong@ln.edu.hk

Abstract

Genetic programming (GP) extends traditional genetic algorithms to automatically induce computer programs. GP has been applied in a wide range of applications such as software re-engineering, electrical circuits synthesis, knowledge engineering, and data mining. One of the most important and challenging research areas in GP is the investigation of ways to successfully evolve recursive programs. A recursive program is one that calls itself either directly or indirectly through other programs. Because recursions lead to compact and general programs and provide a mechanism for reusing program code, they facilitate GP to solve larger and more complicated problems. Nevertheless, it is commonly agreed that the recursive program learning problem is very difficult for GP. In this paper, we propose techniques to tackle the difficulties in learning recursive programs. The techniques are incorporated into an adaptive Grammar Based Genetic Programming system (adaptive GBGP). A number of experiments have been performed to demonstrate that the system improves the effectiveness and efficiency in evolving recursive programs.

Keywords: Grammar Based Genetic Programming, Logic Grammars, Recursive Programs

1. Introduction

Genetic programming (GP) extends traditional genetic algorithms (Holland 1975, Goldberg 1989) to automatically induce computer programs (Koza 1992; 1994, Koza et al. 1999; 2003). It is a stochastic general search and problem solving method that uses the analogies from natural selection and evolution. GP encodes potential solutions to a specific problem as computer programs and apply reproduction and recombination operators to these programs to create new programs. The reproduction and recombination processes are repeated until appropriate solutions are found or all resources have been used. GP has been demonstrated to be effective and robust in searching very large and varied spaces in a wide range of applications such as software re-engineering, electrical circuits synthesis, knowledge engineering (Koza 1992; 1994, Koza et al. 1999, Kinnear 1994, Angeline and Kinnear 1996, Spector et al. 1999), and data mining (Wong and Leung 2000, Freitas 1997).

One of the most important and challenging areas of research in GP 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 reusable representations automatically. Koza (1994) used the Boolean even-n-parity problem to demonstrate extensively that his approach of hierarchical Automatically Defined Functions (ADFs) can facilitate the solving of the problem.

However, GP with ADFs can only solve an instance of the even-n-parity problem for a particular value of n. If a different value of n is provided, GP with ADFs must be used again to induce another program for the new instance of the problem. A better solution is a recursive program that solves all instances of the problem for all $n \geq 0$. A general recursive program is given below:

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

In this recursive program, the argument `L` is a list of Boolean values. Any number of Boolean values can exist in the list `L`.

Since recursive programs are usually compact, elegant, and general solutions of complicated problems, the problem of evolving recursive programs is very important in genetic programming. However, it is commonly agreed that the problem is very difficult.

From our experience in evolving recursive even-n-parity program using Generic Genetic Programming (Wong and Leung 1997), we observed that non-terminating programs with similar structures occur frequently in various generations. In this paper, we propose a technique that automatically modifies the grammar after observing a number of non-terminating programs. The modified grammar reduces the probability of generating this kind of non-terminating programs. We also design a mechanism that changes a grammar to increase/decrease the chance of creating good/bad programs. These techniques accelerate the process of evolving recursive programs.

The techniques are implemented in an adaptive Grammar Based Genetic Programming System (adaptive GBGP), which allows extended logic grammars to be learnt and modified dynamically. The next section describes related research in learning recursive programs. Some difficulties in evolving recursive programs are presented in Section 3. Adaptive GBGP and the techniques of modifying grammars dynamically are discussed in Section 4. The experimental results are presented in the next section (5). In Section 6, we discuss the differences between our approach and other existing methods. The future work and the conclusions are respectively discussed in the last two sections.

2. Related Research

Koza (1992) studied a limited form of recursion for sequence induction. To evolve programs that can generate the Fibonacci sequence, the S-expression was allowed to reference previously computed values in the sequence. Another work on evolving recursive programs is by Brave (1996). GP was applied to evolve programs with recursive ADFs to perform tree search. To evolve a recursive ADF, the name of the ADF was included in its function set. However, an evolved recursive ADF may contain infinite-loops. To handle this problem, the maximum number of recursive calls was specified as the depth of the tree being searched. Usually such a limit affects the evolution process since a good program may never be induced if its evaluation requires more than the permitted recursive calls. It was demonstrated that GP could find solutions to the tree search problem faster than that using non-recursive ADFs. Moreover, the program containing recursive ADFs is less complex and requires less computational effort to execute than the programs with non-recursive ADFs. However, this approach is not a general method to evolve recursive programs.

Whigham designed two directed mutation operators to guide GP to evolve a recursive `member` function using his CFG-GP system (Whigham 1996a). A directed mutation operator specifies that a subtree generated by one particular grammar rule is replaced by another subtree

generated by another grammar rule. However, these two mutation operators are problem specific. The knowledge about the solution is used to direct GP search. For problems that have not an obvious recursive pattern, this approach may not be applicable.

Yu used her PolyGP to evolve `nth` and `map` recursive programs (Yu 1999; 2001a; 2001b, Yu and Clack 1998). In this approach, the name of the program is included in the function set so that it can be used to evolve recursive programs. However, this approach complicates the dynamic of program evolution with other issues. The first issue is the method to handle infinite loops. In her experiments, the maximum number of recursive calls allowed in a program is the length of the input list. This limit may prevent her PolyGP from discovering good programs if the programs require more than the permitted recursive calls to evaluate. The second issue is the fitness penalty applied to programs with infinite loops. It is not clear which fitness penalty is appropriate. Finally, a small change in a recursive program can lead to large variation of the fitness of the program. Thus, recursive programs are extremely deceptive. Therefore, the fitness of a recursive program does not reflect its proximity to a solution in the space of programs.

Yu introduced an alternative approach for evolving recursive programs. In this approach, recursion is provided implicitly by the higher-order function `foldr`. It provides a mechanism of module creation and reuse (Yu 2001a; 2001b).

Recently, Koza and his colleagues introduced Automatically Defined Recursion (ADR) that implements a general form of recursion (Koza et al. 1999). An ADR consists of a Recursion Condition Branch (RCB), a Recursion Body Branch (RBB), a Recursion Update Branch (RUB), and a Recursion Ground Branch (RGB). These branches are subject to evolution during the run of genetic programming. A number of architecture-altering operations for ADR have also been implemented.

Wong and Leung developed a flexible framework called GGP (Generic Genetic Programming). The framework combines GP and Inductive Logic Programming (Lavrac and Dzeroski 1994, Muggleton 1992) to 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 (Wong and Leung 1997, Wong 2001).

Since GGP can induce programs in various programming languages, it must be able to accept grammars of different languages and produce programs in them. Most modern programming languages are specified in the notation of BNF (Backus-Naur Form) that is a kind

of context-free grammar (CFG). However, GGP is based on logic grammars because CFGs (Hopcroft and Ullman 1979) are not expressive enough to represent context-sensitive information of some languages and domain-dependent knowledge of the target programs being induced.

Wong and Leung used GGP to evolve recursive programs for the even-n-parity problem from training examples without noise (Wong and Leung 1996b). Their approach is to construct a logic grammar that includes a grammar rule making recursive calls. Moreover, the grammar enforces a termination condition in the program structure. However, the convergence of recursive calls in the program is not guaranteed. Hence, they used an execution time limit to halt the program. They demonstrated that, using such a grammar to guide evolution, GGP is able to find the solution to the general even-n-parity problem more efficiently than Koza's ADFs approach. They also studied the problem of evolving recursive programs from noisy examples (Wong and Leung 1996a).

Tang et al. (1998) compared Inductive Logic Programming (ILP), GP, and Genetic Logic Programming (GLP is a variant of GP for inducing Prolog programs proposed by Whigham and McKay (1995)) for program induction. These approaches were used to induce four recursive, list-manipulation programs. The results indicate that ILP is generally more accurate at inducing correct programs given limited data and computing resources. GLP performs the worst, and is rarely able to induce a correct program. Although they found that ILP is generally more accurate than GP and GLP, they only used the traditional GP (Koza 1992) in their comparison. Other GP systems such as Strongly Typed GP (Montana 1995), PolyGP, CFG-GP, GGP, and GP with ADR were not compared. Thus, it is not clear if the conclusion is applicable to other GP systems.

3. Difficulties in Evolving Recursive Programs

In general, a recursive program consists of one or more base statements and a number of recursive statements. It is difficult to evolve a recursive program because appropriate base and recursive statements and correct ordering of them must be evolved simultaneously. For example, the following program:

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

is incorrect for the even-n-parity problem, although the second component of the outermost AND function is the target recursive program to be evolved.

Consider the problem of inducing a program from the 8 fitness cases of the even-3-parity problem. Each fitness case is a pair (L_i, Z_i), where $1 \le i \le 8$, L_i is a list of 3 Boolean values and Z_i is a Boolean value. Z_i is True (T) if L_i contains even number of True, otherwise Z_i is False (Nil). The standardized fitness value of an evolved program P is calculated by using the following fitness function:

$$\text{Val(P)} = \sum_{i=1}^{8} \text{MISCLASSIFY}(P, L_i, Z_i)$$

where

$$\text{MISCLASSIFY}(P, L_i, Z_i) = \begin{cases} 1, \text{if P does not terminate for } L_i \\ 1, \text{if P generates run time error for } L_i \\ 1, \text{if P returns T for } L_i \text{ and } Z_i \text{ is Nil} \\ 1, \text{if P returns Nil for } L_i \text{ and } Z_i \text{ is T} \\ 0, \text{otherwise} \end{cases}$$

The following program:

```
(defun parity (L)
  (if (null L) T
    (parity (rest L))))
```

has the standardized fitness value of 4, although its base statement is correct. The standardized fitness value of the program:

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

is 8 (the worst value), although its recursive statement is correct. These examples illustrate that the problem of inducing recursive program is difficult, because the properties of the problem obstruct the construction and combination of good building blocks.

Moreover, several non-terminating programs with similar structures occur frequently in various generations during the evolution of recursive programs. For example, the following programs,

```
(defun parity (L)
  (parity L))

(defun parity (L)
  (AND (parity L) (first L)))

(defun parity (L)
  (OR (parity L) (AND (parity L)
                      (first L))))
```

may be generated several times. Since it is impossible to develop an algorithm that determines if a program will terminate or not, a program is assumed to be non-terminating if it executes for a long time. In other words, much of the execution time is wasted in evaluating these programs, and less execution time is devoted to evolve good programs.

4. Adaptive GBGP

This section presents a novel approach called adaptive GBGP (adaptive Grammar Based Genetic Programming) that is an extention of GGP. Adaptive GBGP applies extended logic grammars to specific the language bias and the search bias of the learning problem of evolving programs (Whigham 1996a; 1996b). This section first introduces the formalism of extended logic grammars followed by the description of the representations and the genetic operators of adaptive GBGP. The techniques of adapting grammars are discussed in Section 4.3.

4.1. Introduction to Extended Logic Grammars

Extended logic grammars are the generalizations of CFGs, which are more expressive than CFGs, but equally amenable to efficient execution. In this paper, extended logic grammars are described in a notation similar to that of definite clause grammars (Pereira and Warren 1980, Pereira and Shieber, 1987; Abramson and Dahl 1989). The grammar for some simple S-expressions in Table 1 will be used throughout this section.

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

Table 1: An extended logic grammar

An extended logic grammar differs from a CFG in that the 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 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.

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 `(- 2.0 2.5)` of a program if `?x` and `?y` are instantiated respectively to 2.0 and 2.5. 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.

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`. In another example, 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. In Table 1, 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 underlined number before each rule is used to identify this rule.

The difference between an extended logic grammar and a logic grammar is that the former allows a non-terminal at the right hand side of a grammar rule to be followed by an optional list of *rule-biases*. A rule-biases list is enclosed by a pair of angle brackets and it contains a list of pairs. The first element of a pair is a number that identifies a grammar rule while the second element of a pair is an integer between *min-rule-bias* and *max-rule-bias*. In the current implementation, min-rule-bias and max-rule-bias are respectively 0 and 5. The second element is called *rule-bias* and it specifies the relative probability of applying the corresponding grammar rule to expand the non-terminal symbol. Since a terminal symbol cannot be expanded by applying grammar rules, it should not have a rule-biases list.

For example, consider the first non-terminal symbol `exp-1(?x)` of grammar rule 2, its rule-biases list is <(5 2) (6 2) (7 1)>, thus the probabilities of applying grammar rules

5, 6, and 7 to expand the non-terminal symbol are respectively 0.4, 0.4 and 0.2. If the rule-biases list of a non-terminal symbol is not specified, an initial value (which is 3 in this implementation) is assigned to the rule-bias of every applicable grammar rules. Therefore, the rule-biases list of the second non-terminal symbol `exp-1(?x)` of grammar rule 2 is `<(5 3) (6 3) (7 3)>`. In other words, the probabilities of applying grammar rules 5, 6, and 7 to expand this non-terminal symbol are equal. A similar (but much simpler) approach was described by Whigham (1996a).

4.2. Representations, Crossover, and Mutation

Adaptive GBGP represents a program as a derivation tree showing how the program has been derived from the extended logic grammar. In other words, a derivation tree is the genotype and the corresponding program is the phenotype. Adaptive GBGP applies deduction to randomly generate programs and their derivation trees in the language declared by the given grammar. These derivation trees form the initial population and adaptive GBGP directly manipulates these trees to find appropriate solutions. For example, the program `(* (+ X 0) (+ X 0))` can be generated by adaptive GBGP given the extended logic grammar in Table 1. Its derivation tree is depicted in Figure 1. The bindings of logic variables are shown in italic font and enclosed in a pair of braces.

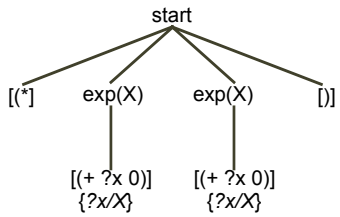


Figure 1: The derivation tree of the program `(* (+ X 0) (+ X 0))`.

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 operation takes place on terminal and/or non-terminal nodes of the two parental derivation trees and produces one offspring program. The crossover algorithm is given as follows:

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.

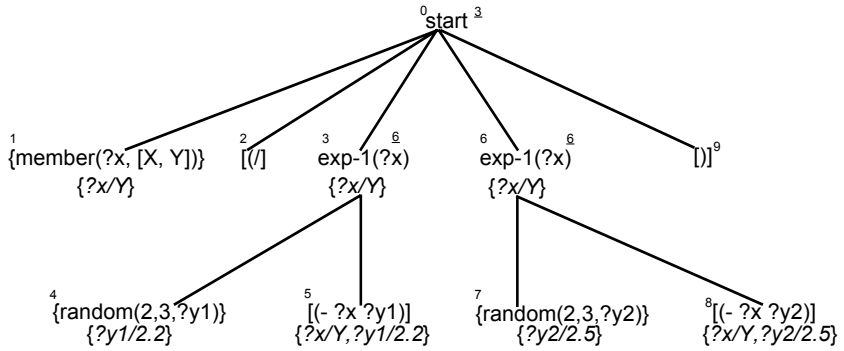
The root of the selected sub-tree is called the primary crossover point. Otherwise, terminate the algorithm without generating any offspring.

2. Select another sub-tree in the secondary derivation tree using a distribution based on the information maintained in the rule-biases list of different non-terminal symbols. The selected sub-tree fulfills the constraint that the offspring 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.

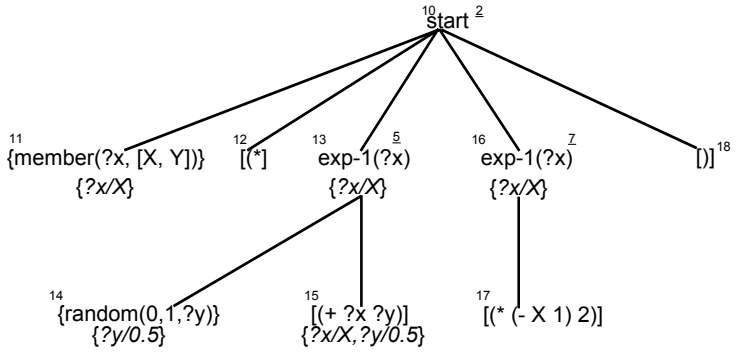
Consider two parental programs generated by the grammar in Table 1, the primary program is `(/ (- Y 2.2) (- Y 2.5))` and the secondary program is `(* (+ X 0.5) (* (- X 1) 2))`. The corresponding derivation trees are depicted in Figures 2(a) and 2(b) respectively. In the figures, the plain numbers identify sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in parsing the corresponding sub-trees.

For example, the primary and secondary sub-trees are 3 and 13 respectively. The valid offspring `(/ (+ Y 0.5) (- Y 2.5))` is produced and the derivation tree is shown in Figure 3. 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 22 is instantiated to Y in sub-tree 20. This example demonstrates the use of extended logic grammars to enforce contextual-dependency between different constituents of a program.

Adaptive GBGP 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.



(a)



(b)

Figure 2: **Derivation trees of the parental programs: a) The derivations tree of the primary parental program `(/ (- Y 2.2) (- Y 2.5))`; b) The derivations tree of the secondary parental program `(* (+ X 0.5) (* (- X 1) 2))`.**

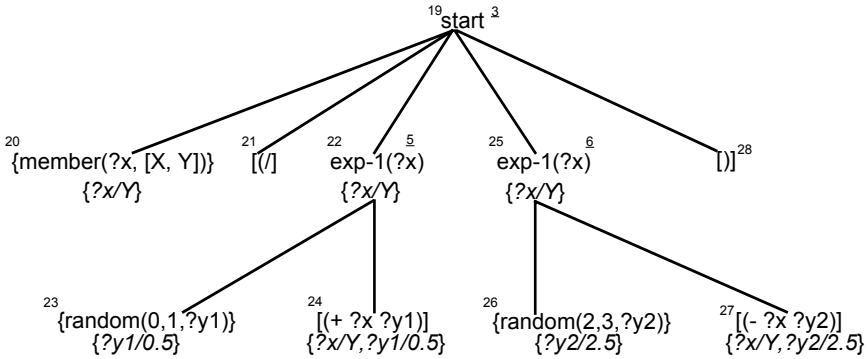


Figure 3: **A derivation tree of the offspring produced by performing crossover between the primary sub-tree 3 of the tree in Figure 2(a) and the secondary sub-tree 13 of the tree in Figure 2(b).**

The mutation operation introduces random modifications to programs in the population. A program in the population is selected as the parental program. The following steps are used to produce an offspring program:

1. If there are sub-trees in the derivation tree of the parental program 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 mutation point. Otherwise, terminate the algorithm without generating any offspring.
2. Generate a new derivation tree by using the information maintained in the rule-biases list of different non-terminal symbols. The tree fulfills the constraints specified by the grammar.
3. If a new derivation-tree can be found in step 2, create and return the offspring, which is obtained by deleting the selected sub-tree and then inserting the new derivation tree at the mutation point. Otherwise, go to step 1.

For example, assume that the program being mutated is `(/ (- Y 2.2) (- Y 2.5))` and the corresponding derivation tree is depicted in Figure 2(a). If the sub-tree 3, MUTATED-SUB-TREE, is selected to be modified and the root of the MUTATED-SUB-TREE is designated as the MUTATE-POINT. Then a new derivation tree, NEW-SUB-TREE, for the S-expression `(+ Y 0.9)` can be obtained from the non-terminal symbol `exp-1(Y)` using the fifth rule of the grammar. The derivation tree is shown in Figure 4. A new offspring is obtained by duplicating the genetic materials of its parental derivation tree, followed by deleting the MUTATED-SUB-TREE from the duplication, and then pasting the NEW-SUB-TREE at the MUTATE-POINT. The derivation tree of the offspring `(/ (+ Y 0.9) (- Y 2.5))` can be found in Figure 5.

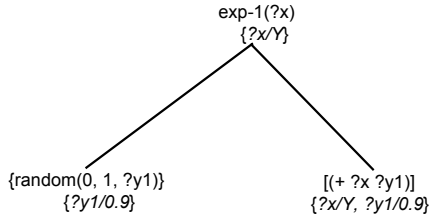


Figure 4: A derivation tree generated from the non-terminal `exp-1(Y)`.

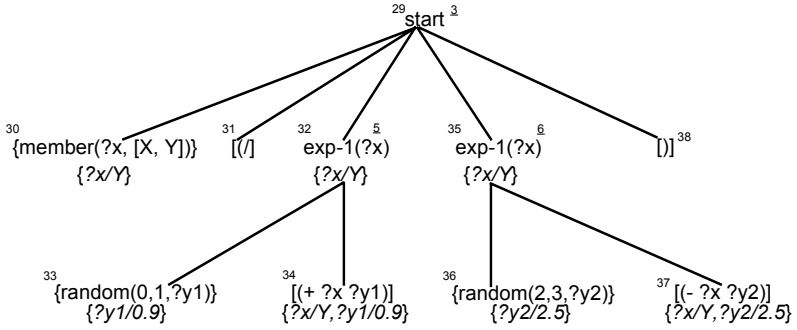


Figure 5: A derivation tree of the offspring produced by performing mutation of the tree in Figure 2(a) at the sub-tree 3.

4.3. Adaptations of Extended Logic Grammars

Two techniques are implemented in adaptive GBGP, which allows extended logic grammars to be modified dynamically when offspring are created by crossover operation. The first one attempts to increase/decrease the probability of generating good/poor programs. The second one tries to reduce the chance of creating non-terminating programs. It should be emphasized that similar techniques have not been developed for mutation operation.

4.3.1. First Adaptive technique

Consider the grammar depicted in Table 2 for the even-n-parity problem,

```
11:start              ->    [ (defun parity (L) ], s-expr(BOOL), [ ) ].
12:s-expr(BOOL)       ->    [ T ].
13:s-expr(BOOL)       ->    [ nil ].
14:s-expr(BOOL)       ->    [ ( ], op, s-expr(BOOL), s-expr(BOOL), [ ) ].
15:s-expr(BOOL)       ->    [ ( ], [ if (null ], s-expr(LIST), [ ) ],
                           s-expr(BOOL), s-expr(BOOL), [ ) ].
16:s-expr(BOOL)       ->    [ ( ], [ parity ], s-expr(LIST)<(18 3)(19 3)>,
                           [ ) ].
17:s-expr(BOOL)       ->    [ ( ], [ first ], s-expr(LIST)<(18 3)(19 3)>,
                           [ ) ].
18:s-expr(LIST)       ->    [ L ].
19:s-expr(LIST)       ->    [ ( ],[ rest ], s-expr(LIST),[ ) ].
20:op                 ->    [ AND ].
21:op                 ->    [ OR ].
22:op                 ->    [ NAND ].
23:op                 ->    [ NOR ].
```

Table 2: An extended logic grammar for the even-n-parity problem.

this grammar allows the program,

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

and the program,

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

to be generated. The derivation trees of these two programs are shown in Figures 6 and 7 respectively. If these programs are used to classify all fitness cases of the even-3-parity problem, their standardized fitness values are 3. If the first and the second programs are respectively selected as the primary and the secondary parents for crossover, an offspring,

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

will be created if the sub-trees 39 and 42 are exchanged. Since the fitness value of the offspring is 0, which is better than its parents, we should increase the chance of performing this kind of crossover.

From the derivation tree of the offspring shown in Figure 8, it can be observed that the grammar rule 18 is used to deduce the sub-tree 45 (a copy of the sub-tree 42) and the grammar rule 17 (the rule number at the sub-tree 44) contains the non-terminal symbol `s-expr(LIST)`, this situation indicates that the probability of applying the grammar rule 18 to deduce the sub-tree for the non-terminal symbol `s-expr(LIST)` of the grammar rule 17 should be increased, in order to increase the chance of performing this kind of crossover.

In another example, the same primary and secondary parents are used. An offspring,

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

will be created if the sub-trees 41 and 43 are exchanged. Since the fitness value of the offspring is 4, which is worse than its parents, we should decrease the chance of executing this kind of crossover. From the derivation tree of the offspring shown in Figure 9, it can be observed that the grammar rule 19 is used to deduce the sub-tree 47 (a copy of the sub-tree 43) and the grammar rule 17 (the rule number at the sub-tree 46) contains the non-terminal symbol `s-expr(LIST)`, this situation indicates that the probability of using the grammar rule 19 to deduce the sub-tree for the non-terminal symbol `s-expr(LIST)` of the grammar rule 17 should be decrease.

In order to avoid the problem of changing a grammar based on the performance of only one offspring, adaptive GBGP examines the performance of a number of offspring generated by similar crossover operations and determines how to modify the grammar. For each offspring created by crossover that terminates, adaptive GBGP invokes the algorithm shown in Table 3. Firstly, the algorithm checks if the offspring should be used to change the grammar (line 4). If this is the case, the algorithm finds the grammar rule that should be altered, the list of rule-biases, and the pair (rule number and rule-bias) corresponds to the rule number at the crossover point (lines 5 – 8).

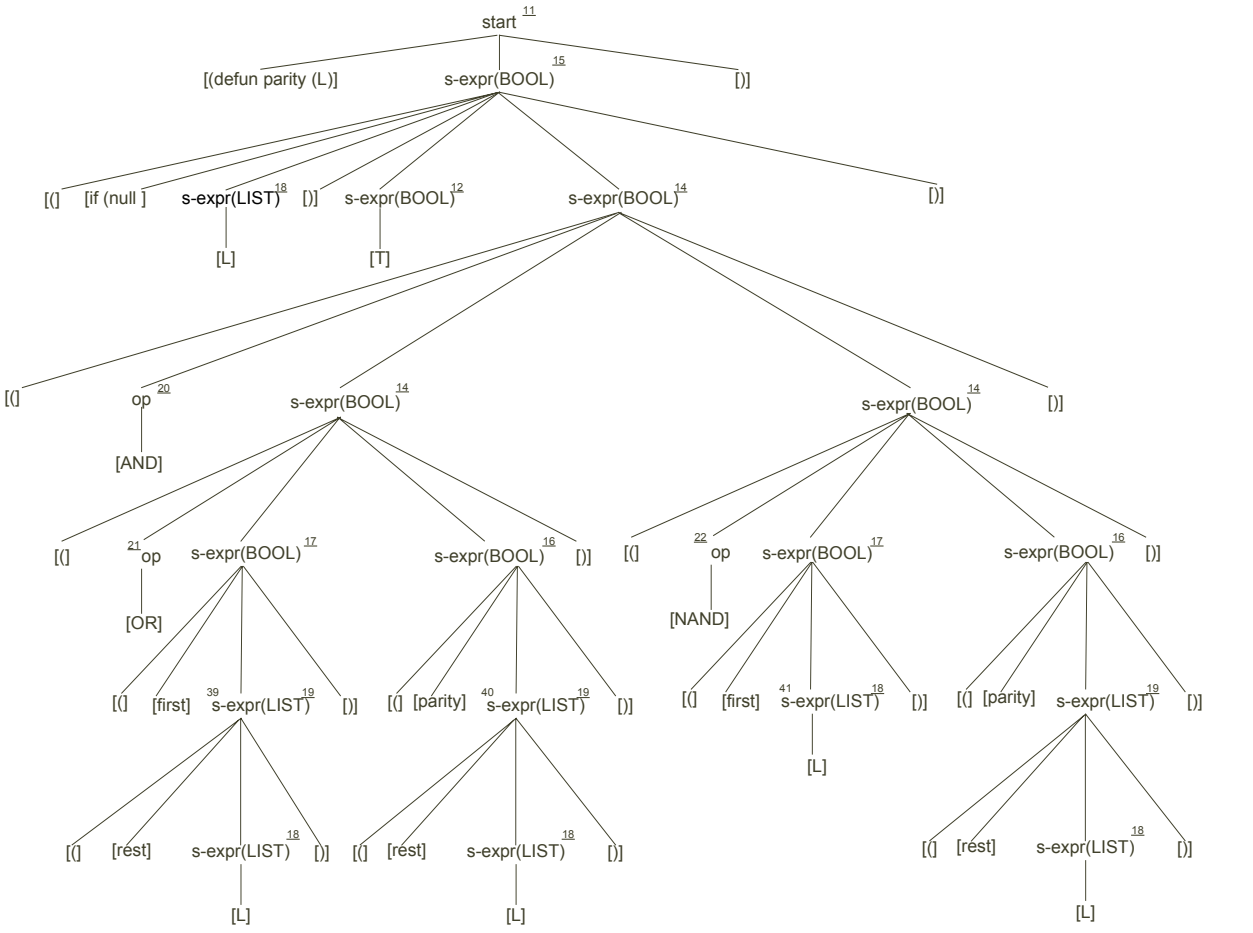


Figure 6: Derivation tree of the primary parent `(defun parity (L) (if (null L) T (AND (OR (first (rest L)) (parity (rest L))) (NAND (first L) (parity (rest L))))))`. Its fitness value is 3 for the even-3-parity problem.

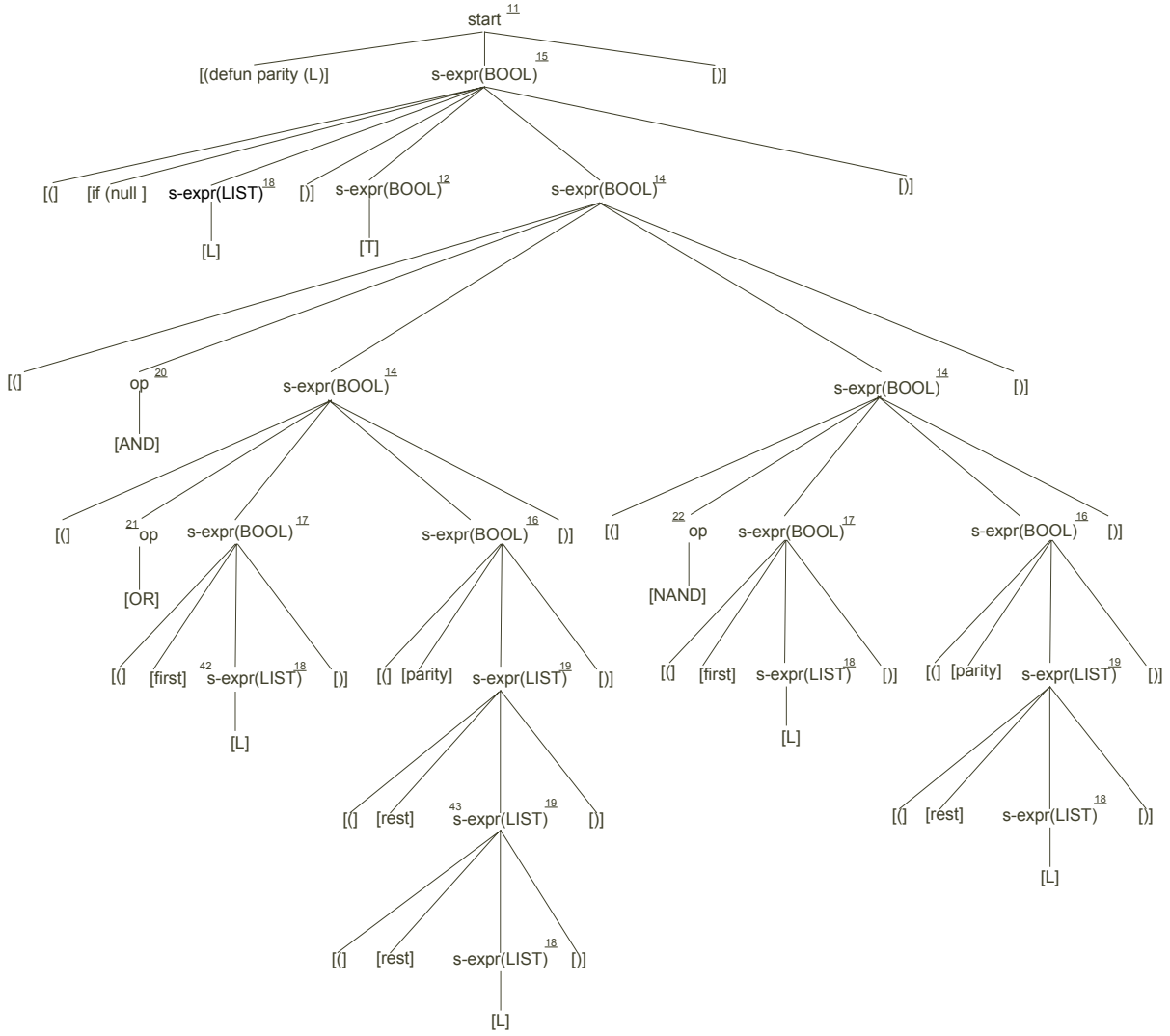


Figure 7: Derivation tree of the secondary parent `(defun parity (L) (if (null L) T (AND (OR (first L) (parity (rest (rest L)))) (NAND (first L) (parity (rest L))))))`. Its fitness value is 3 for the even-3-parity problem.

Each pair is associated with a 4-tuple that summarizes the performance of the offspring produced by a particular kind of crossover operations. The 4-tuple contains `dec`, n_d, `inc`, and n_i, which are respectively the cumulative decrement of standardized fitness value of the offspring, the number of the offspring which are better than their parents, the cumulative increment of standardized fitness value of the offspring, and the number of the offspring which are worse than their parents. They are initialized to zero at the beginning of the evolution. From line 11 to line 15, the values of the 4-tuple are updated.

If the total number of the offspring having better or worse performance reaches a specific value α, the information in the 4-tuple will be used to determine how to alter the rule-bias in the pair (lines 16 – 21). The rule-bias will be increased by 1 if the average decrement of standardized fitness value is larger than the average increment of standardized fitness values (lines 17 – 18). On the other hand, the rule-bias will be decreased if the average decrement of standardized fitness value is smaller than the average increment of standardized fitness values (lines 19 – 20). When modifying the rule-bias, the algorithm ensures that the rule-bias will not be smaller than min-rule-bias or greater than max-rule-bias.

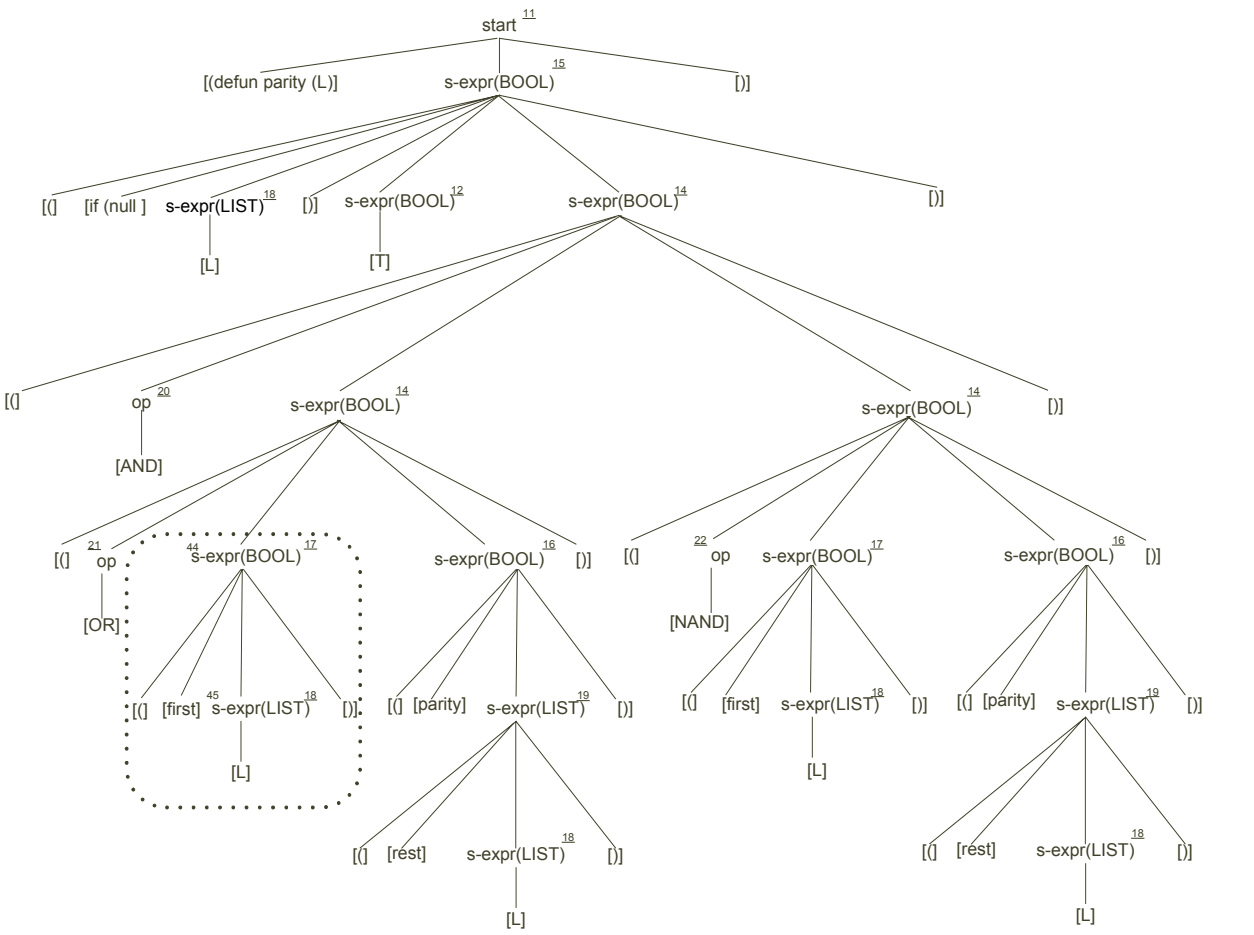


Figure 8: A derivation tree of the offspring produced by performing crossover between the sub-tree 39 of the tree in Figure 6 and the sub-tree 42 of the tree in Figure 7. Its fitness value is 0 for the even-3-parity problem. The rounded rectangle highlights the crossover point and its parent node.

For example, consider the offspring depicted in Figure 8, the crossover point is at the sub-tree 45, the fitness value of the offspring is 0, and the fitness values of its parents are 3. The

condition at line 4 is satisfied because the grammar symbol at the sub-tree 45, `s-expr(LIST)`, is a non-terminal and the offspring is better than its parents. Thus the following statements (lines 5 – 21) are executed. The rule number at the crossover point, R_c, is 18 and the rule number at the parent node (the sub-tree 44) of the crossover point, R_p, is 17. The list of rule-biases, `Rule-biases-list`, associated with `s-expr(LIST)` is `<(18 3) (19 3)>` and the pair with the rule number 18, `Pair`, is `(18 3)`. Assume that all values of the 4-tuple associated with `Pair` are 0, the values of `dec` and n_d are changed to 3 and 1 respectively (lines 12 – 13). Assume that the value of α is 1, the conditions at lines 16 and 17 are satisfied and the rule-bias in `Pair` is increased by 1. Consequently, the grammar rule 17 is changed to,

```
s-expr(BOOL)   ->   [ ( ], [ first ], s-expr(LIST)<(18 4)(19 3)>, [ ) ].
```

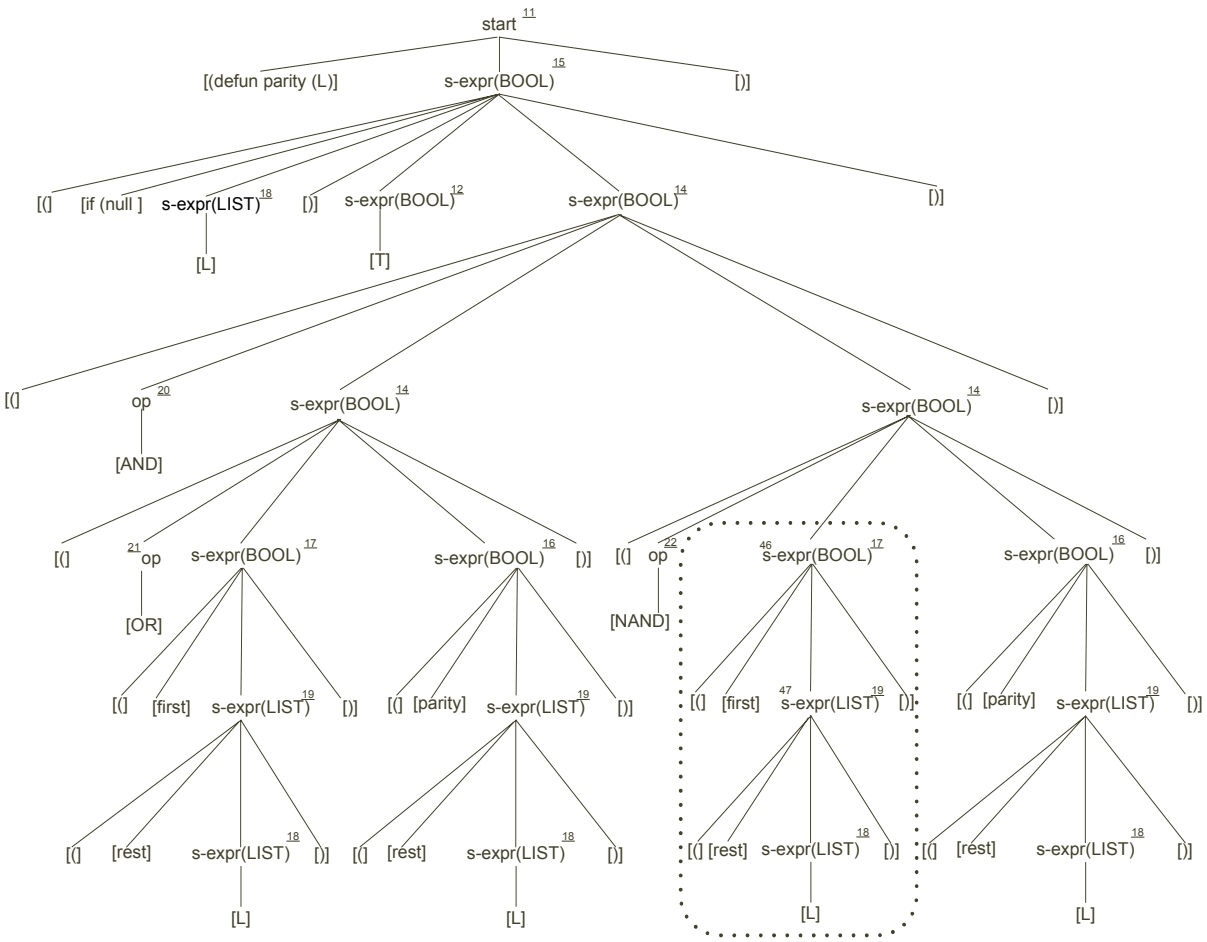


Figure 9: **A derivation tree of the offspring produced by performing crossover between the sub-tree 41 of the tree in Figure 6 and the sub-tree 43 of the tree in Figure 7. Its fitness value is 4 for the even-3-parity problem. The rounded rectangle highlights the crossover point and its parent node.**

```
Input:
  P:     The primary derivation tree.
  S:     The secondary derivation tree.
  O:     The derivation tree of the offspring.
  C:     The crossover point in the offspring.
/* This algorithm assumes that better programs have smaller
   standardized fitness values                                          */
Procedure modify-rule-bias(P, S, O, C)
{
  f_p := The fitness value of P;                                    /* 1 */
  f_s := The fitness value of S;                                    /* 2 */
  f_o := The fitness value of O;                                    /* 3 */
  if ((the grammar symbol at C is a non-terminal) and
      ((f_o < min(f_p, f_s)) or (f_o > max(f_p, f_s)))) {           /* 4 */
    R_c := The grammar rule number at C;                            /* 5 */
    R_p := The grammar rule number at the parent node of C;         /* 6 */
    Rule := The grammar rule with rule number R_p;                  /* 7 */
    Rule-biases-list := the list of rule-biases associated with
                        the non-terminal symbol which is expanded
                        to the crossovered sub-tree;                /* 8 */
    Pair := The pair in Rule-biases-list with R_c;                  /* 9 */
    (dec, n_d, inc, n_i) := The 4-tuple associated with Pair;       /* 10 */
    if (f_o < min(f_p, f_s)) {                                      /* 11 */
      /* The offspring is better than its parent */
      dec := dec + (min(f_p, f_s) - f_o);                           /* 12 */
      n_d := n_d + 1;                                               /* 13 */
    }
    else {
      /* The offspring is worse than its parent */
      inc := inc + (f_o - max(f_p, f_s));                           /* 14 */
      n_i := n_i + 1;                                               /* 15 */
    }
    if ((n_d + n_i) >= α) {                                         /* 16 */
      if ((((n_i = 0) and (n_d > 0)) or
           ((n_i > 0) and (n_d > 0) and (dec/n_d > inc/n_i))) and
          (the rule-bias in Pair < max-rule-bias))                  /* 17 */
        Increase the rule-bias in Pair by 1;                        /* 18 */
      if ((((n_d = 0) and (n_i > 0)) or
           ((n_d > 0) and (n_i > 0) and (dec/n_d < inc/n_i))) and
          (the rule-bias in Pair > min-rule-bias))                  /* 19 */
        Decrease the rule-bias in Pair by 1;                        /* 20 */
      Set dec, n_d, inc, n_i to 0;                                  /* 21 */
    }
  }
}
```

Table 3: The algorithm that modifies an extended logic grammar to increase/decrease the probability of generating good/poor programs.

4.3.2. Second Adaptive technique

From our experience in evolving recursive even-n-parity programs using GGP, we have observed that non-terminating programs with similar structures occur frequently in various generations. Consider the primary and the secondary programs depicted in Figures 6 and 7, a non-terminating program,

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

will be created if the sub-trees 40 and 42 are exchanged. From the derivation tree of the offspring shown in Figure 10, it can be observed that the grammar rule 18 is used to deduce the sub-tree 49 (a copy of sub-tree 42) and the grammar rule 16 (the rule number at the sub-tree 48) contains the non-terminal symbol `s-expr(LIST)`, this situation indicates that the probability of applying the grammar rule 18 to deduce the sub-tree for the non-terminal symbol `s-expr(LIST)` of the grammar rule 16 should be reduced, in order to decrease the chance of generating similar non-terminating programs. The algorithm that realizes this idea is given in Table 4. In addition to non-terminating programs, the algorithm also considers the information from other programs created by similar crossover operations, so as to prevent the problem of modifying a rule-bias incorrectly.

Adaptive GBGP calls the algorithm to process each offspring created by crossover. If the grammar symbol at the crossover point is a non-terminal (line 1), the algorithm finds the grammar rule that should be altered, the list of rule-biases, and the pair (rule number and rule-bias) corresponds to the rule number at the crossover point (lines 2 – 5). Each pair is associated with two values, $F_{\text{non-terminating}}$ and $U_{\text{bias-modification}}$. The first one is a Boolean value that identifies if non-terminating programs have been generated by crossover at this location. It is initialized to false. Since the rule-bias can not be greater than max-rule-bias nor smaller than min-rule-bias, the second value specifies the number of rule-bias modifications that have not been processed yet. If $U_{\text{bias-modification}}$ is smaller than 0, it indicates that some non-terminating programs have been produced but their effects have not been reflected in the rule-bias yet. On the other hand, if it is greater than 0, the effects of some programs that can terminate have not been revealed in the rule-bias. $U_{\text{bias-modification}}$ is initialized to 0 at the beginning of evolution.

If the offspring does not terminate (line 7), the algorithm decreases the value of $U_{\text{bias-modification}}$ (line 10) or the rule-bias in the pair (line 11) depending on different conditions (line 9). On the other hand, if the offspring does terminate and other non-terminating programs have been produced previously at this location (line 12), the algorithm increases the value of $U_{\text{bias-modification}}$ (line 14) or the rule-bias in the pair (line 15).

For example, consider the offspring depicted in Figure 10 with the crossover point at the sub-tree 49, the condition at line 1 is satisfied because the corresponding grammar symbol, `s-expr(LIST)`, is a non-terminal. Thus the following statements (lines 2 – 15) are executed. The

rule number at the crossover point, R_c, is 18 and the rule number at the parent node (the sub-tree 48) of the crossover point, R_p, is 16. The list of rule-biases, `Rule-biases-list`, associated with `s-expr(LIST)` is <(18 3) (19 3)> and the pair with the rule number 18, `Pair`, is (18 3). Assume that $F_{non\text{-}terminating}$ and $U_{bias\text{-}modification}$ associated with `Pair` are false and 0, respectively, $F_{non\text{-}terminating}$ is changed to true (line 8) and the rule-bias in `Pair` is reduced (line 11). Consequently, the grammar rule 16 is updated to,

```
s-expr(BOOL)    ->    [ ( ], [ parity ], s-expr(LIST)<(18 2)(19 3)>, [ ) ].
```

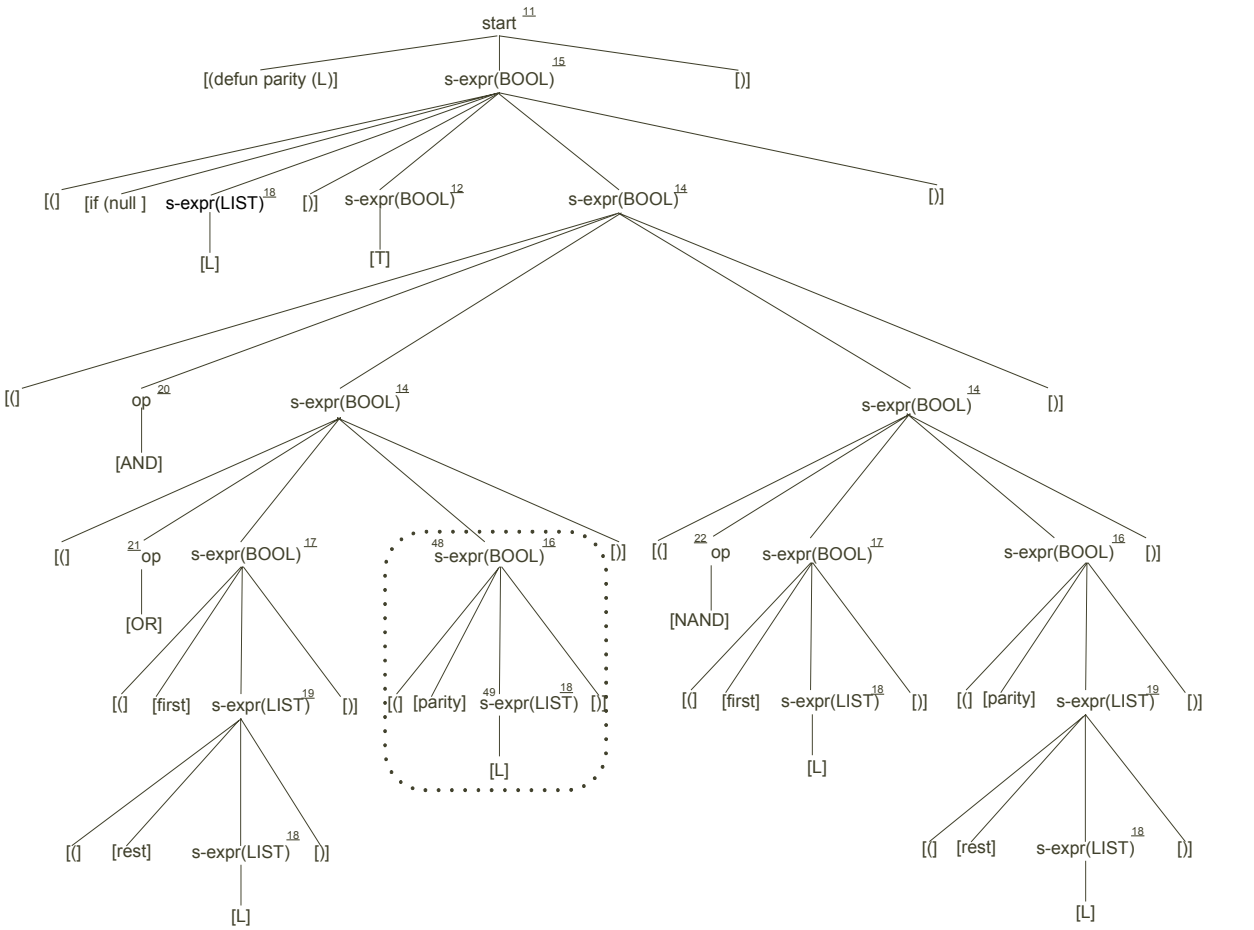


Figure 10: A derivation tree of the offspring produced by performing crossover between the sub-tree 40 of the tree in Figure 6 and the sub-tree 42 of the tree in Figure 7. The rounded rectangle highlights the crossover point and its parent node.

Similar, if the same offspring are created twice at the same crossover point, the grammar rule 16 is changed to,

```
s-expr(BOOL)    ->    [ ( ], [ parity ], s-expr(LIST)<(18 0)(19 3)>, [ ) ].
```

At this time, if the same offspring is produced again at the same crossover point, the condition at line 9 is satisfied because the rule-bias in `Pair` is equal to min-rule-bias, which is 0 in our case. Consequently, $U_{bias\text{-}modification}$ is updated to –1 (line 10) to indicate that a non-terminating program has been found, but the rule-bias in `Pair` has not been modified accordingly because it is invalid to set the value of the rule-bias to –1.

```
Input:
  O:     The derivation tree of the offspring.
  C:     The crossover point in the offspring.

Procedure modify-rule-bias-non-terminating(O, C)
{
  if (the grammar symbol at C is a non-terminal) {              /* 1 */
    Rc := The grammar rule number at C;                         /* 2 */
    Rp := The grammar rule number at the parent node of C;      /* 3 */
    Rule := The grammar rule with rule number Rp;               /* 4 */
    Rule-biases-list := the list of rule-biases associated with
                        the non-terminal symbol which is expanded
                        to the crossovered sub-tree;            /* 5 */
    Pair := The pair in Rule-biases-list with Rc;               /* 6 */
    if (O is non-terminating) {                                 /* 7 */
      /* Fnon-terminating specifies if non-terminating programs have been
         generated by similar crossover operations  */
      Set the Fnon-terminating associated with Pair to True;    /* 8 */
      if ((the Ubias-modification associated with Pair > 0) or
          (the rule-bias in Pair <= min-rule-bias))             /* 9 */
        decrease the Ubias-modification associated with Pair by 1;   /* 10 */
      else
        decrease the rule-bias in Pair by 1;                    /* 11 */
    }
    else if (the Fnon-terminating associated with Pair is True) {   /* 12 */
      if ((the Ubias-modification associated with Pair < 0) or
          (the rule-bias in Pair >= max-rule-bias))             /* 13 */
        increase the Ubias-modification associated with Pair by 1;   /* 14 */
      else
        increase the rule-bias in Pair by 1;                    /* 15 */
  }
}
```

Table 4: The algorithm that modifies an extended logic grammar to decrease the probability of generating non-terminating programs.

5. Experiments

In Sections 5.1, 5.2, and 5.3, we study the performance of adaptive GBGP for the 11-multiplexer, the even-n-parity and the greater-all problems. The grammars modified by adaptive GBGP are examined in Section 5.4 using a new problem. All systems used in these experiments are generation-based GP.

5.1. The Boolean 11-Multiplexer Problem

We compare the performance of adaptive GBGP and non-adaptive GBGP on the Boolean 11-multiplexer problem. Since recursive programs are not produced in this problem, it can be used to evaluate the effectiveness of the first adaptive mechanism described in Section 4.3.1. In this experiment, adaptive GBGP and non-adaptive GBGP apply the extended logic grammar in Table 5 to evolve the function that takes 3 address bits and 2^3 data bits as its input. The value returned by the multiplexer function is the Boolean value (true or false) of the particular data bit that is selected by the 3 address bits of the multiplexer.

The parameters of different systems are listed in Table 6. Ten percent of the offspring are generated by the replacement operation proposed by Whigham (1996a). At each generation, the replacement operator produces new offspring by using the extended logic grammar, which is varying in adaptive GBGP and is invariable in non-adaptive GBGP. The training set contains all 2048 fitness cases from the 11-multiplexer problem. The standardized fitness value of an evolved program is the total number of misclassifications on the 2048 fitness cases. The evolution terminates if the maximum number of generations of 50 is reached or a program that classifies all fitness cases correctly is found. For adaptive GBGP, the value of α in Table 3 is 5.

```
31:start        ->    [ (defun mult (a0 a1 a2 d0 d1 d2 d3 d4 d5 d6 d7) ],
                      s-expr, [ ) ].
32:s-expr       ->    [ ( ], op, s-expr, s-expr, [ ) ].
33:s-expr       ->    [ (NOT ], s-expr, [ ) ].
34:s-expr       ->    [ (IF ], s-expr, s-expr, s-expr, [ ) ].
35:s-expr       ->    Data.
36:s-expr       ->    Address.
37:op           ->    [ AND ].
38:op           ->    [ OR ].
39:Data         ->    [ d0 ].
40:Data         ->    [ d1 ].
41:Data         ->    [ d2 ].
42:Data         ->    [ d3 ].
43:Data         ->    [ d4 ].
44:Data         ->    [ d5 ].
45:Data         ->    [ d6 ].
46:Data         ->    [ d7 ].
47:Address      ->    [ a0 ].
48:Address      ->    [ a1 ].
49:Address      ->    [ a2 ].
```

Table 5: An extended logic grammar for the 11-multiplexer problem.

The experiment is repeated for 1,000 times. Adaptive GBGP and non-adaptive GBGP respectively evolve 20 and 10 programs that classify all fitness cases correctly. The difference is statistically significant at the 0.05 level using a one-tailed test for the difference between two

population proportions. The curves in Figure 11 show the experimentally observed cumulative frequency of success of solving the problem by generation.

Parameter	Value
Population Size	500
Maximum Number of Generations	50
Crossover Rate	0.6
Mutation Rate	0.2
Replacement Rate	0.1
Rate of offspring that are copied directly from parents	0.1
Maximum Tree Depth at The Initial Population	8
Maximum Tree Depth after Crossover or Mutation	10

Table 6: Parameters for the 11-multiplexer problem.

The curves in Figure 12 show the number of programs I(M, i, z) that must be processed to produce a solution by generation i with a probability z, where M is 1,000 and z is 0.99. The I(M, i, z) of adaptive GBGP reaches a minimum value of 5,584,731 at generation 48 (Koza 1992). On the other hand, the I(M, i, z) of non-adaptive GBGP reaches a minimum value of 11,226,160 at generation 48. From Figures 11 and 12, it can be observed that adaptive GBGP performs significantly better than non-adaptive GBGP. Thus, the first adaptive mechanism is effective in improving the performance of the system.

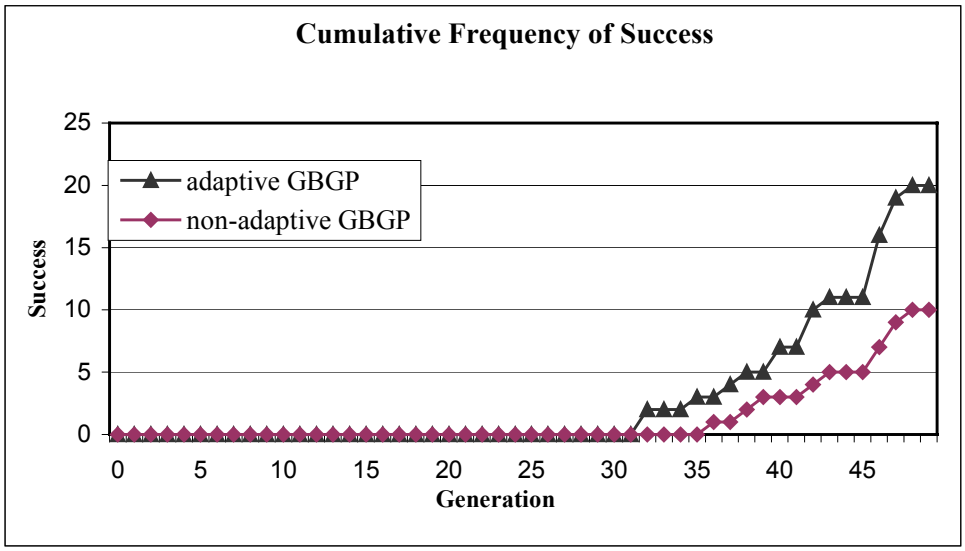


Figure 11: The performance curves showing cumulative frequency of success for the 11-multiplexer problem.

It should be noted that Koza's GP and CFP-GP could not solve the 11-multiplexer problem with a population size of 500 and the maximum number of generations of 50 (Koza 1992, Whigham 1996a).

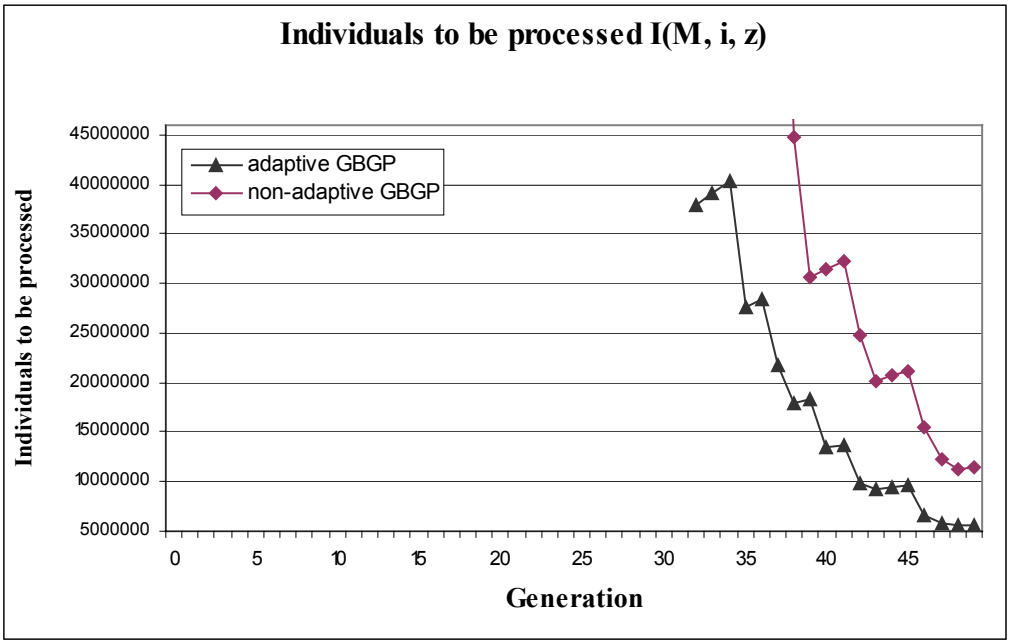


Figure 12: The performance curves showing I(M, i, z) for the the 11-multiplexer problem. M is 1,000 and z is 0.99.

5.2. The Even-n-Parity Problem

In this experiment, we compare the effectiveness of adaptive GBGP, adaptive GBGP2, and non-adaptive GBGP that use the extended logic grammar in Table 2 to evolve recursive programs for the even-n-parity problem. In adaptive GBGP2, the first adaptive technique is disabled so that we can evaluate the performance of the mechanism that reduces the chance of generating non-terminating programs.

The parameters of different systems for this problem are listed in Table 7. 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 program is the total number of misclassifications on the 13 fitness cases. The evolution terminates if the maximum number of generations of 100 is reached or a program that classifies all fitness cases correctly is obtained.

In order to avoid the problem caused by a non-terminating recursive program, a recursion limit is enforced. After invoking the program recursively for 20 times, if the evolved program fails to find a result for a fitness case, it will be terminated. In this case, the program is assumed

to be non-terminating and a special fitness value is assigned to it to indicate that it is non-terminating. It is possible that an evolved program will generate exceptions during its execution for some fitness cases. For example, it is illegal to perform the `first` operation on an empty list. If the program produces an exception, it is assumed that it will misclassify the corresponding fitness cases.

Parameter	Value
Population Size	500
Maximum Number of Generations	100
Crossover Rate	0.6
Mutation Rate	0.2
Replacement Rate	0.1
Rate of offspring that are copied directly from parents	0.1
Maximum Tree Depth at The Initial Population	8
Maximum Tree Depth after Crossover or Mutation	12

Table 7: Parameters for the even-n-parity problem.

In Section 5.2.1, we present the experimental results of non-adaptive GBGP. The performance results of adaptive GBGP2 and adaptive GBGP are described in Sections 5.2.2 and 5.2.3, respectively. One of the grammars modified by adaptive GBGP is discussed in Section 5.2.4. All results are obtained from 500 runs of the experiment.

5.2.1. Non-adaptive GBGP

Non-adaptive GBGP successfully evolves 73 programs that classify all fitness cases correctly. The generated programs are then tested on the even-i-parity problems, where $i \in \{0, 1, 2, 4, 5, 6, 7, 8, 9, 10\}$. 61 of them can successfully solve all the problems. They are further analyzed manually and it is found that these 61 programs are correct recursive programs for the general even-n-parity problem. The experimentally observed cumulative frequency of success is depicted in Figure 13.

From the curve in Figure 14, it can be observed that the minimum value of I(M, i, z) is 1,283,857 at generation 58. Since there are 13 fitness cases, 1,283,857 *13 = 16,690,141 fitness cases should be processed to find a general recursive program for the problem. The average number of non-terminating programs produced by non-adaptive GBGP is 2,018.00.

5.2.2. Adaptive GBGP2

In the 500 trials, adaptive GBGP2 successfully evolves 83 programs that classify all fitness cases correctly. The generated programs are further studied and 82 of them are correct recursive programs. From Figure 13, it can be found that adaptive GBGP2 performs better than non-adaptive GBGP. The difference is statistically significant at the 0.05 level using a one-tailed test for the difference between two population proportions.

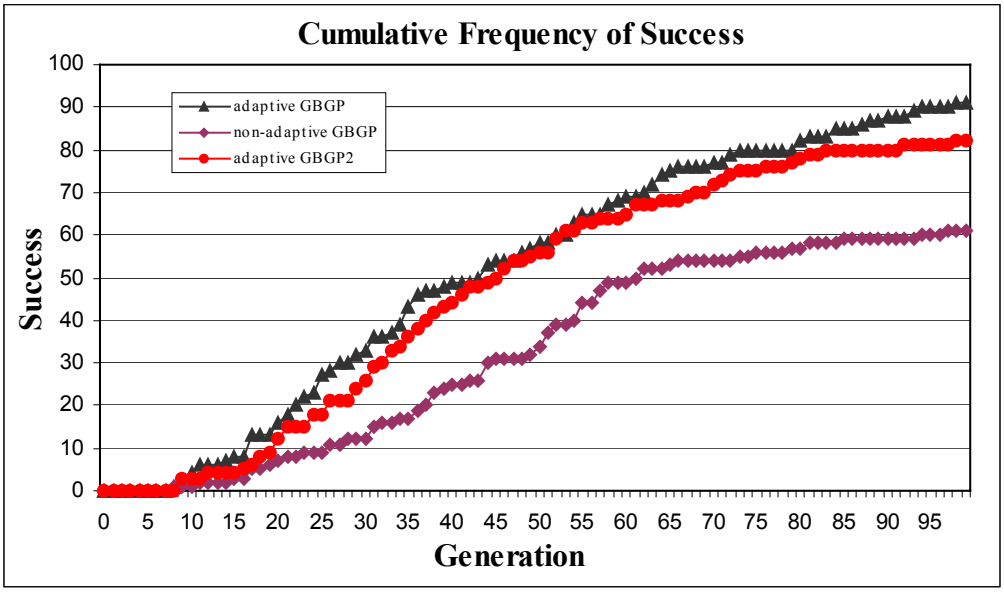


Figure 13: The performance curves showing cumulative frequency of success for the even-n-parity problem.

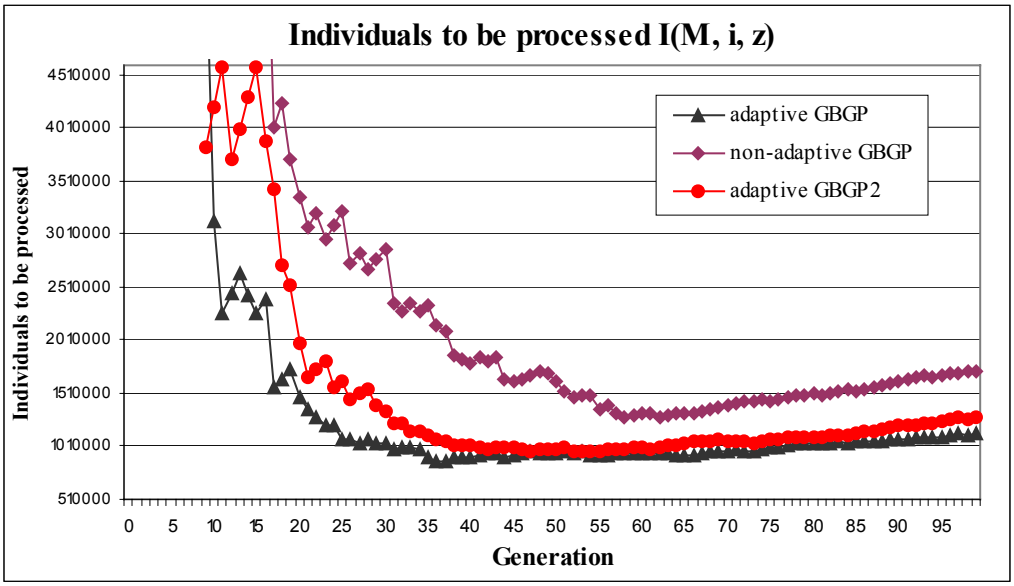


Figure 14: The performance curves showing I(M, i, z) for the even-n-parity problem. M is 500 and z is 0.99.

From Figure 14, the I(M, i, z) of adaptive GBGP2 reaches a minimum value of 953,619 at generation 53, which is much smaller than that of non-adaptive GBGP. The average number of non-terminating programs produced by adaptive GBGP2 is 939.2, which is significantly smaller than that of non-adaptive GBGP at 0.05 level using a one-tailed *t*-test. Consequently, the second adaptive mechanism can reduce the chance of producing non-terminating programs.

5.2.3. Adaptive GBGP

Adaptive GBGP successfully evolves 91 correct recursive programs for the general even-n-parity problem. From Figure 13, it can be seen that adaptive GBGP performs better than the other two systems. The difference between adaptive GBGP and non-adaptive GBGP is statistically significant at the 0.05 level. The I(M, i, z) of adaptive GBGP reaches a minimum value of 873,490 at generation 36, which is lower than those of non-adaptive GBGP and adaptive GBGP2 (Figure 14). Since there are only 13 fitness cases, 873,490*13 = 11,355,370 fitness cases should be processed to find a general recursive program for the even-n-parity problem.

On the other hand, GP with ADFs generates about 1,440,000 programs to obtain at least one solution with 99% probability for the even-7-parity problem (Koza 1994). Since there are 2^7 fitness cases, GP with ADFs evaluates $1{,}440{,}000*2^7 = 184{,}320{,}000$ fitness cases to find a program that solves the even-7-parity problem only. In other words, adaptive GBGP can solve the even-7-parity problem about 16 times faster.

Yu applied her PolyGP to solve the even-n-parity problem with a success rate of 80% (Yu 2001a; 2001b). The I(M, i, z) of PolyGP reaches a minimum value of 17,500 at generation 4. Since there are only 12 fitness cases in her experiments, 17,500*12 = 210,000 fitness cases should be processed. However, the performance of PolyGP and adaptive GBGP should not be compared directly because PolyGP uses a higher-order function `foldr`, which is very useful in evolving recursive programs.

The average number of non-terminating programs produced by adaptive GBGP is 933.26, which is smaller than those of non-adaptive GBGP and adaptive GBGP2. Moreover, the difference between adaptive GBGP and non-adaptive GBGP is significant at 0.05 level using a one-tailed *t*-test.

This experiment demonstrates that adaptive GBGP has highest success rate, lowest minimum value of I(M, i, z), and smallest number of non-terminating programs generated, thus

the two adaptive mechanisms are effectively for solving the problem of learning recursive programs.

The average execution time of non-adaptive GBGP, adaptive GBGP2, and adaptive GBGP are respectively 1,087.59 seconds, 1,161.16 seconds, and 1,077.83 seconds[1]. The average execution time of different systems is not significantly different at 0.05 level. In other words, the two adaptive mechanisms do not introduce significant overhead.

5.2.4. Modified Grammar

One of the extended logic grammars modified by adaptive GBGP is shown in Table 8.

```
11:start              ->    [ (defun parity (L) ), s-expr(BOOL)
                            <(12 5)(13 5)(14 4)(15 5)(16 4)(17 5)>, [ ) ].
12:s-expr(BOOL)       ->    [ T ].
13:s-expr(BOOL)       ->    [ nil ].
14:s-expr(BOOL)       ->    [ ( ], op <(20 4)(21 4)(22 1)(23 5)>,
                            s-expr(BOOL)
                            <(12 3)(13 4)(14 3)(15 4)(16 4)(17 5)>,
                            s-expr(BOOL)
                            <(12 3)(13 5)(14 4)(15 4)(16 4)(17 4)>, [ ) ].
15:s-expr(BOOL)       ->    [ ( ], [ if (null ], s-expr(LIST)
                            <(18 5)(19 3)>, [ ) ], s-expr(BOOL)
                            <(12 5)(13 4)(14 4)(15 4)(16 4)(17 4)>,
                            s-expr(BOOL)
                            <(12 4)(13 5)(14 4)(15 5)(16 4)(17 4)>, [ ) ].
16:s-expr(BOOL)       ->    [ ( ], [ parity ], s-expr(LIST)<(18 0)(19 3)>,
                            [ ) ].
17:s-expr(BOOL)       ->    [ ( ], [ first ], s-expr(LIST)<(18 1)(19 3)>,
                            [ ) ].
18:s-expr(LIST)       ->    [ L ].
19:s-expr(LIST)       ->    [ ( ],[rest], s-expr(LIST)<(18 5)(19 3)>,[ ) ].
20:op                 ->    [ AND ].
21:op                 ->    [ OR ].
22:op                 ->    [ NAND ].
23:op                 ->    [ NOR ].
```

Table 8: An extended logic grammar altered by adaptive GBGP. The modified grammar rules are shaded.

The grammar disallows some non-terminating programs such as,

```
(defun parity (L)
 (AND (first L)
      (parity L)))
```

to be evolved and thus it accelerates the process of evolving recursive programs.

We perform another experiment that applies non-adaptive GBGP with this grammar. Ninety-three correct recursive programs are learnt and the I(M, i, z) reaches a minimum value of

[1] The systems are executed on the same Linux machine with a Pentium IV 1.4 GHz CPU and 1GB memory.

846,927 at generation 38. The average, maximum, and minimum numbers of non-terminating programs produced by non-adaptive GBGP are 47.18, 402, and 0 respectively. In other words, the modified grammar leads to higher success rate, reduced value of I(M, i, z), and fewer non-terminating programs being evolved. This experiment suggests that the two adaptive mechanisms can capture the characteristics of the even-n-parity problem and encode this knowledge in the grammar so that the learning process can be accelerated if the same problem is attempted again. In Section 5.4, we will show that the modified grammar is also beneficial for another problem of learning recursive programs.

5.3. The Greater-All Problem

Adaptive GBGP and non-adaptive GBGP are further compared on the greater-all problem to demonstrate that the two adaptive mechanisms are applicable for other problems of learning recursive programs. The two systems apply the extended logic grammar in Table 9 to evolve a function that takes two input parameters. The first one is a number and the second one is a list of numbers. The function returns true if the first number is greater than all numbers in the second list, otherwise it returns false.

```
51:start              ->    [ (defun greater-all (x L) ], s-expr(BOOL),
                            [ ) ].
52:s-expr(BOOL)       ->    [ T ].
53:s-expr(BOOL)       ->    [ nil ].
54:s-expr(BOOL)       ->    [ ( ], op, s-expr(BOOL), s-expr(BOOL), [ ) ].
55:s-expr(BOOL)       ->    [ ( ], op2, s-expr(NUM), s-expr(NUM), [ ) ].
56:s-expr(BOOL)       ->    [ ( null ], s-expr(LIST), [ ) ].
57:s-expr(BOOL)       ->    [ ( ], [ if ], s-expr(BOOL), s-expr(BOOL),
                           s-expr(BOOL), [ ) ].
58:s-expr(BOOL)       ->    [ ( ], [ greater-all x ], s-expr(LIST), [ ) ].
59:s-expr(NUM)        ->    [ ( ], [ first ], s-expr(LIST), [ ) ].
60:s-expr(NUM)        ->    [ x ].
61:s-expr(LIST)       ->    [ L ].
62:s-expr(LIST)       ->    [ ( ],[ rest ], s-expr(LIST),[ ) ].
63:op                 ->    [ AND ].
64:op                 ->    [ OR ].
65:op2                ->    [ > ].
66:op2                ->    [ < ].
67:op2                ->    [ = ].
```

Table 9: An extended logic grammar for the greater-all problem.

The parameters of different systems are summarized in Table 10. The training set contains 26 fitness cases and 10 of them are positive examples. The standardized fitness value of an evolved program is the total number of misclassifications on the 26 fitness cases. The evolution terminates if the maximum number of generations of 50 is reached or a program that

classifies all fitness cases correctly is found. A program is assumed to be non-terminating if it is invoked recursively for 50 times. All results are obtained from 100 runs of the experiment.

In 100 trials, non-adaptive GBGP successfully evolves 75 programs that classify all fitness cases correctly. They are analyzed manually and it is found that 69 of them are correct recursive programs for the greater-all problem. The experimentally observed cumulative frequency of success is depicted in Figure 15.

Parameter	Value
Population Size	500
Maximum Number of Generations	50
Crossover Rate	0.6
Mutation Rate	0.3
Replacement Rate	0.0
Rate of offspring that are copied directly from parents	0.1
Maximum Tree Depth at The Initial Population	8
Maximum Tree Depth after Crossover or Mutation	10

Table 10: Parameters for the greater-all problem.

From the curve in Figure 16, the minimum value of I(M, i, z) is 63,828 at generation 22. Since there are 26 fitness cases, 63,828 *26 = 1,659,528 fitness cases should be processed to find a general recursive program for the problem. The average number of non-terminating programs produced by non-adaptive GBGP is 368.00.

Adaptive GBGP successfully evolves 84 correct recursive programs for the greater-all problem. From Figure 15, it can be seen that adaptive GBGP performs better than the other one. The difference is statistically significant at the 0.05 level. The I(M, i, z) of adaptive GBGP reaches a minimum value of 44,915 at generation 18, which is lower than that of non-adaptive GBGP (Figure 16). Since there are only 26 fitness cases, 44,915*26 = 1,167,790 fitness cases should be processed to find a general recursive program for the problem. The average number of non-terminating programs produced by adaptive GBGP is 321.45, which is slightly smaller than that of non-adaptive GBGP.

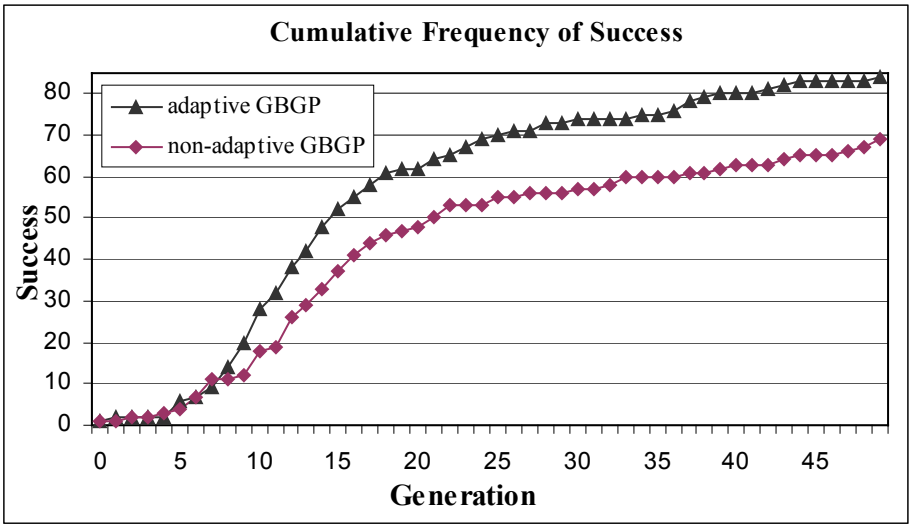


Figure 15: The performance curves showing cumulative frequency of success for the greater-all problem.

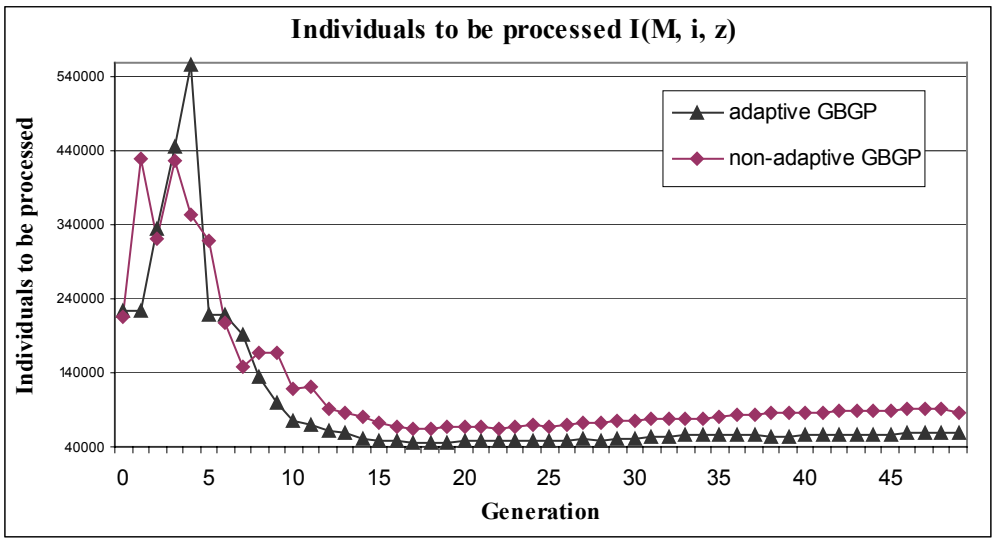


Figure 16: The performance curves showing I(M, i, z) for the greater-all problem. M is 500 and z is 0.99.

5.4. The Modified Parity Problem

To study if the grammars learnt by adaptive GBGP in Section 5.2.3 can be used for other problems, we apply non-adaptive GBGP with different grammars to evolve recursive programs for a problem modified from the even-n-parity problem. The problem aims at learning a recursive program that takes a list with n Boolean values and considers only the values at the first, the third, the fifth,…, locations. The program returns true if an even number of the considered values are true, otherwise it returns false.

Firstly, we compare non-adaptive GBGP with grammars in Tables 2 and 8. The parameters for this experiment are summarized in Table 11. The replacement operation is not executed because grammars are not adapted in this experiment. The modified-0-, 1-, 2-, 3-, 4-, and 5-parity problems are used in the training process. The training set contains all 63 fitness cases from these problems. The standardized fitness value of an evolved program is the total number of misclassifications on the 63 fitness cases. The evolution terminates if the maximum number of generations of 100 is reached or a program that classifies all fitness cases correctly is found. All results are obtained from 100 runs of the experiment.

Parameter	**Value**
Population Size	500
Maximum Number of Generations	100
Crossover Rate	0.6
Mutation Rate	0.3
Replacement Rate	0.0
Rate of offspring that are copied directly from parents	0.1
Maximum Tree Depth at The Initial Population	8
Maximum Tree Depth after Crossover or Mutation	12

Table 11: Parameters for the modified-n-parity problem.

In 100 trials, non-adaptive GBGP with the original grammar (Table 2) evolves only one program that classifies all fitness cases correctly. It is then tested on the modified-i-parity problems, where $i \in \{0, 1, 2, 4, 5, 6, 7, 8, 9, 10\}$. It can successfully solve all the problems. It is further analyzed manually and we confirm that it is a correct recursive program for the general modified-n-parity problem. The curves in Figure 17 show the experimentally observed cumulative frequency of success of solving the problem by generation. The I(M, i, z) reaches a minimum value of 16,495,581 at generation 71 (Figure 18). Since there are 63 fitness cases, 16,495,581*63 = 1,039,221,603 fitness cases should be processed to find a general recursive program for the problem. The average number of non-terminating programs produced by non-adaptive GBGP is 1563.49. By comparing these results with those described in Section 5.2.1, we can conclude that the modified-n-parity problem is much more difficult than the even-n-parity problem.

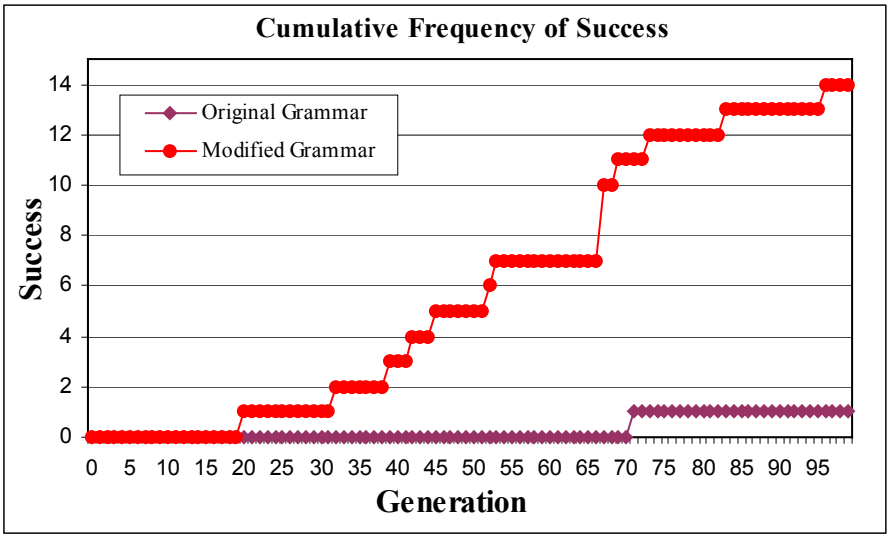


Figure 17: The performance curves showing cumulative frequency of success for the modified-n-parity problem.

On the other hand, non-adaptive GBGP with the modified grammar in Table 8 evolves 14 correct recursive programs for the problem. The success rate of the modified grammar is significantly higher than that of the original grammar at 0.05 level. The I(M, i, z) reaches a minimum value of 1,332,918 at generation 73 (Figure 18). Thus, a general recursive program for the problem can be found if 1,332,918*63 = 83,973,834 fitness cases are processed. The average number of non-terminating programs generated is 64.05.

This experiment shows that non-adaptive GBGP with the modified grammar has higher success rate, lower minimum of I(M, I, z), and smaller number of non-terminating programs generated, thus the modified grammar is better than the original grammar.

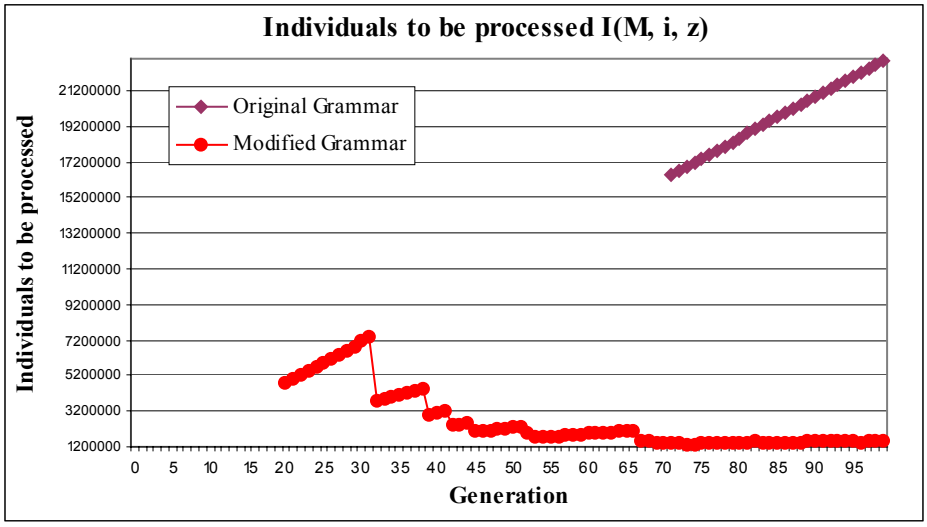


Figure 18: The performance curves showing I(M, i, z) for the modified-n-parity problem. M is 500 and z is 0.99.

In additional to the modified grammar listed in Table 8, other grammars obtained in the 91 successful runs in Section 5.2.3 are also used by non-adaptive GBGP to learn recursive programs for the problem. The parameters used in this experiment are summarized in Table 11. The experimental results for non-adaptive GBGP with the original grammar and the first 10 adapted grammars are summarized in Table 12[2]. Numbers in parentheses are the standard deviations. The success rates of the adapted grammars are significantly higher than that of the original grammar at 0.05 level and the numbers of non-terminating programs created by the adapted grammars are significantly smaller than that of the original grammar at 0.05 level. This experiment suggests that the knowledge discovered by adaptive GBGP for the even-n-parity problem can also be used for a different but similar problem.

Grammar	Success Rate (100 trials)	Minimum of I(M, i, z)	Number of Non-Terminating Programs		
			Average	Maximum	Minimum
1	0.14	1,332,918	64.05 (82.30)	360	1
2	0.11	1,538,139	118.31 (140.41)	911	8
3	0.15	1,159,831	101.18 (129.56)	663	2
4	0.08	2,009,516	61.50 (95.87)	633	2
5	0.19	1,016,228	95.16 (170.16)	1,504	1
6	0.13	1,521,146	39.06 (54.16)	242	1
7	0.11	1,738,788	82.15 (148.75)	1,208	1
8	0.09	1,794,977	54.33 (94.47)	606	0
9	0.15	1,099,211	54.36 (70.90)	522	0
10	0.13	1,332,918	121.20 (201.48)	1,640	4
Original	0.01	16,495,581	1,563.49 (321.17)	2,616	976

Table 12: Experimental results of non-adaptive GBGP with different grammars for the modified-n-parity problem.

We perform another experiment to determine if the original grammar (Table 2) and the modified grammar (Table 8) have sufficient biases for generating correct recursive programs without crossover and mutation. In this experiment, 500,000,000 programs are created randomly and no correct recursive programs can be obtained for the two grammars. This experiment suggests that genetic operators are essential for this problem.

[2] The remaining 81 adapted grammars have similar performance. The success rates of these grammars are significantly higher than that of the original grammar at 0.05 level and the numbers of non-terminating programs created by these grammars are significantly smaller than that of the original grammar at 0.05 level.

6. Discussion

Whigham developed a framework for automatically modifying an initial context-free grammar in his CFG-GP system. The technique improved the convergence of CFG-GP for the 6-multiplexer problem (Whigham 1996a). However, he did not demonstrate if this technique can be used in evolving recursive programs. His approach has a number of characteristics. Firstly, new grammar rules can be added to the grammar but existing rules cannot be deleted. Secondly, the modified grammars must represent the same language that is expressible from the initial grammar. Thirdly, new grammar rules are extracted from the fittest program in each generation. Grammar rules cannot be obtained from useful derivations in other programs, and thus useful information may be wasted. Finally, the approach assumes that any terminal in the fittest program may contribute to developing useful grammar rules. Thus, the learnt grammar rules only specify the structures of the lower part of the derivation tree. Wong and Leung (1996b) demonstrated that grammar rules describing the overall structure of the derivation tree are very useful in evolving recursive programs. But the approach of Whigham cannot learn this kind of grammar rules.

Although our adaptive GBGP will not explicitly delete any existing rules, the effect of removing rules can be achieved by using extended logic grammars. If all occurrences of the rule-bias for a specific rule are 0, the rule is effectively deleted. Thus, the modified grammars may represent different languages. Moreover, our adaptive GBGP modifies a grammar by considering performance information from a number of offspring, thus it is less likely to change a grammar inappropriately. Extended logic grammars also allow the probabilities of applying rules to be different in various contexts (i.e. rules). Since the same non-terminal symbol at the right-hand side of different rules can have different rule-biases list, rules may have different probabilities of being used in different contexts. For example, consider the grammar rule 2 in Table 1, the probabilities of applying rules 5, 6, and 7 to expand the first non-terminal symbol `exp-1(?x)` are 0.4, 0.4, and 0.2, respectively. On the other hand, the probabilities of using rules 5, 6, and 7 to expand the first non-terminal symbol `exp-1(?x)` of the grammar rule 3 are 0.6, 0.2, and 0.2, respectively.

Angeline (1996) used adaptive techniques for determining crossover position with GP. For each program tree, a parameter tree having the same structure as the program is maintained. At each node in the parameter tree, there is a value that represents the probability of performing crossover at that node. These values are adaptively modified using a Gaussian random noise

after each crossover operation. Instead, our adaptive GBGP maintains this kind of information in the grammar and it is updated by examining the fitness values of the offspring created by crossover.

Recently, O'Neill and Ryan proposed Grammatical Evolution by Grammatical Evolution $(GE)^2$ that has two distinct grammars, the *universal grammar* and the *solution grammar* (O'Neill and Ryan 2004). Each individual in $(GE)^2$ has two chromosomes. The first one is in the *universal grammar* that is employed to specify the *solution grammar* being used. The second chromosome applies the *solution grammar* to specify the solution. O'Neill and Ryan demonstrated the feasibility of the evolution of grammatical evolution's grammar on a number of symbolic regression problems. The results illustrated the ability of learning the importance of various terminal symbols. It is interesting to study if $(GE)^2$ can be used for learning the biases of rules in the *solution grammar* and for handling recursive problems.

Shan et al. (2004) developed Grammar Model-based Program Evolution (GMPE) that employs a probabilistic context-free grammar to model promising programs. A stochastic hill-climbing learning algorithm is used to generalize the initial grammar given by the users. The grammar model can better represent, preserve, and promote the building blocks in good individuals. Shan et al. demonstrated that GMPE significantly outperforms traditional GP on the Royal Tree problem and the Max problem.

7. Future Work

Since the recursion limit may prevent different GBGP systems from discovering good programs if the programs require more than the allowed recursive calls to evaluate, the recursion limit may influence the conclusions drawn on the experiments done in Sections 5.2, 5.3, and 5.4. Systematic experiments will be done to determine the effect of the recursion limit on the performance in evolving recursive programs.

To determine if the technique described in this paper is general enough to handle various extended logic grammars and different problems, we will apply adaptive GBGP on a number of recursive program learning problems including factorial, Fibonacci, member, reverse, conc, last, shift, and translate functions with and without noisy and missing training examples.

We will study methods to add new rules and delete existing rules dynamically. The modified grammars should represent different languages. Thus, if the initial grammar does not contain the solution, the modified grammars may allow adaptive GBGP to find the solution. We will also investigate if the adaptive techniques can be applied in other grammar based genetic

programming systems such as adaptive logic programming (Keijzer and Babovic 2002, Keijzer et al. 2001), grammatical evolution (O'Neill and Ryan 2001) and grammatical evolution by grammatical evolution (O'Neill and Ryan 2004).

8. Conclusion

In this paper, we have proposed techniques to tackle the difficulties in learning recursive programs by dynamically modifying the grammar specifying the search space. The modified grammar increase/decrease the probability of generating good/bad offspring and reduce the chance of producing non-terminating programs, thus it accelerates the process of evolving recursive programs. The techniques are incorporated into an adaptive Grammar Based Genetic Programming system (adaptive GBGP). A number of experiments have been performed to demonstrate that the system improves the effectiveness and efficiency in evolving recursive programs.

Acknowledgements

This research was supported by the Earmarked Grant LU 3009/02E from the Research Grant Council of the Hong Kong Special Administrative.

Reference

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

Angeline, P. J. (1996). Two Self-Adaptive Crossover Operators for Genetic Programming. In In P. J. Angeline and K. E. Kinnear, Jr. (editors). *Advances in Genetic Programming 2*. pp. 89 - 109. MA: MIT Press.

Angeline, P. J. and Kinnear, K. E. Jr. editors (1996). *Advances in Genetic Programming 2*. MA: MIT Press.

Brave, S. (1996). Evolving Recursive Programs for Tree Search. In P. J. Angeline and K. E. Kinnear, Jr. (editors). *Advances in Genetic Programming 2*. pp. 203- 219. MA: MIT Press.

Freitas, A. A. (1997). A Genetic Programming Framework for two Data Mining Tasks: Classification and Generalized Rule Induction. In *Genetic Programming 1997: Proceedings of the 2nd Annual Conference*, pp. 96-101.

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

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

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

Keijzer, M. and Babovic, V., (2002) Declarative and Preferential Bias in GP-based Scientific Discovery. *Genetic Programming and Evolvable Machines*, **3**, pp. 41 - 79.

Keijzer, M., Babovic, V., Ryan, C, O'Neill, M., and Cattolico, M. (2001) Adaptive Logic Programming. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H. –M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke (editors). *Proceedings of the Genetic and Evolutionary Computation Conference 2001*, pp. 42 - 49. CA: Morgan Kaufmann.

Kinnear, K. E. Jr. editors (1994). *Advances in Genetic Programming 1*. MA: MIT Press.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Boston, MI: Kluwer Academic Publishers.

Koza, J. R., Bennett, F. H. III, Andre, D., and Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann Publishers.

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

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

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

Montana, D. J. (1995). Strongly Typed Genetic Programming. *Evolutionary Computation*, **3**, pp. 199-230.

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

O'Neill, M. and Ryan, C. (2004) Grammatical Evolution by Grammatical Evolution: The Evolution of Grammar and Genetic Code. In *Proceedings of the Seventh European Conference on Genetic Programming*, pp138-149.

O'Neill, M. and Ryan, C. (2001) Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*, **5**, pp. 349 - 358.

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

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.

Shan, Y., McKay, R. I., Baxter, R., Abbass, H., Essam, D., Nguyen, H. X. (2004). Grammar Model-Based Program Evolution. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pp. 478-485.

Spector, L., Langdon, W. B., O'Reilly, U. M., and Angeline, P. J. editors (1999). *Advances in Genetic Programming 3*. MA: MIT Press.

Tang, L, R, Califf, M. E., and Mooney, R. J. (1998) *An Experimental Comparison of Genetic Programming and Inductive Logic Programming on Learning Recursive List Functions*. TR AI98-271, Artificial Intelligence Lab, University of Texas at Austin.

Whigham, P. A. (1996a*). Grammatical Bias for Evolutionary Learning*. Ph.D. Thesis. University of New South Wales.

Whigham, P. A. (1996b). Search Bias, Language Bias, and Genetic Programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference***.** pp. 230-237. MA: MIT Press.

Whigham, P. A. and McKay, R. (1995) Genetic Approaches to learning recursive relations. In *Lecture Notes in Artificial Intelligence: Volume 956*. pp. 17-28.

Wong, M. L. (2001). A Flexible Knowledge Discovery System using Genetic Programming and Logic Grammars. *Decision Support Systems*, **31**, pp. 405-428.

Wong, M. L. and Leung, K. S. (2000). *Data Mining Using Grammar Based Genetic Programming and the Applications*. Boston, MA: Kluwer Academic Publishers.

Wong, M. L. and Leung, K. S. (1997). Evolutionary Program Induction Directed by Logic Grammars. *Evolutionary Computation*, **5**, pp. 143-180.

Wong, M. L. and Leung, K. S. (1996a). Learning Recursive Functions from Noisy Examples using Generic Genetic Programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference***.** pp. 238-246. MA: MIT Press.

Wong, M. L. and Leung, K. S. (1996b). Evolving Recursive Functions for the Even-parity Problem Using Genetic Programming. In P. J. Angeline and K. E. Kinnear, Jr. (editors). *Advances in Genetic Programming 2*. pp. 221- 240. MA: MIT Press.

Yu, T. (2001a). Hierarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lambda Abstraction. *Genetic Programming and Evolvable Machines*, **2**, pp. 345-380.

Yu, T. (2001b). Polymorphism and Genetic Programming. In *Proceedings of the Fourth European Conference on Genetic Programming*, pp. 416-421.

Yu, T. (1999). *An analysis of the Impact of Functional Programming Techniques on Genetic Programming*. Ph.D. Thesis. Department of Computer Science. University College London.

Yu, T., and Clack, C. (1998). PolyGP: A Polymorphic Genetic Programming System in Haskell. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (editors). *Genetic Programming 1998: Proceedings of the Third Annual Conference***.** pp. 416-421. CA: Morgan Kaufmann.