Applying Adaptive Grammar Based Genetic Programming in Evolving Recursive Programs

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 a technique to tackle the difficulties in learning recursive programs. The technique is incorporated into an adaptive Grammar Based Genetic Programming system (adaptive GBGP). A number of experiments have been performed to demonstrate that the system can evolve recursive programs efficiently and effectively.

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). 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 genetic programming is the investigation of ways to apply them to larger and more complicated problems. One approach to make a large problem more tractable is to discover problem representations automatically. Koza (1994) 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.

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

Koza showed that the even-7-parity problem can be solved using GP with ADFs. He found that about 1,440,000 programs should be evaluated to obtain at least one solution with 99% probability. Since all 2^n fitness cases, for a particular value of n, were used as the training examples, it is unclear whether GP with ADFs can discover the regularities of the even-n-parity problem and induce a general program. Moreover, 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, and thus it accelerates the process of evolving recursive programs.

The technique is 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 technique of modifying grammars dynamically are discussed in Section 4. The experiment results are presented in the next section. In Section 6, we discuss the differences between our approach and other existing methods. A conclusion is given in the last section.

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.

GP was applied to evolve programs with recursive ADFs to perform tree search (Brave 1996). 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 can 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 1996). 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 1999b). 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 1999a).

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) which 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 a recursive program 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. Consider the even-n-parity problem, the following program:

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

is incorrect, although the second component of the outermost AND function is the target recursive program to be evolved.

Moreover, consider the problem of inducing a program from all fitness cases of the even-3-parity problem, the standardized fitness value of the program:

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

is only 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 Grammar Based Genetic Programming system (adaptive GBGP) 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 1996). 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 technique of adapting grammars is discussed in Section 4.3.

4.1 Introduction to Extended Logic Grammars

Extended logic grammars are the generalizations of CFGs. Their expressiveness is much more powerful than those of 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). The grammar for some simple S-expressions in Table 1 will be used throughout this section.

```
1:start ->  [(*], exp(W), exp(W), exp(W), [)].
2:start ->  {member(?x,[W, Z])}, [(*],
            exp-1(?x) <(5 2)(6 2)(7 1)>,
            exp-1(?x), exp-1(?x),[)].
3:start ->  {member(?x,[W, Z])}, [(+),
            exp-1(?x) <(5 3)(6 1)(7 1)>,
            exp-1(?x), exp-1(?x), [)].
4:exp(?x)   ->  [(/ ?x 1.5)].
5:exp-1(?x) -> {random(1,2,?y)}, [(/ ?x ?y)].
6:exp-1(?x) -> {random(3,4,?y)}, [(- ?x ?y)].
7:exp-1(W)  ->  [(+ (- W 11) 12)].
```

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 `(- 1.0 2.0)` of a program if `?x` and `?y` are instantiated respectively to 1.0 and 2.0. Non-terminal symbols are similar to literals in Prolog; `exp-1(?x)` in Table 1 is an example of non-terminal symbol. Commas denote concatenation and each grammar rule ends with a full stop.

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, [W, Z])` in Table 1 instantiates the variable `?x` to either `W` or `Z` if `?x` has not been instantiated, otherwise it checks whether the value of `?x` is either `W` or `Z`. In another example, if the variable `?y` has not been bound, the goal `random(1, 2, ?y)` instantiates `?y` to a random floating point number between 1 and 2. Otherwise, the goal checks whether the value of `?y` is between 1 and 2.

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.

One of the differences 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 1 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. 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, the maximum value (i.e. max-rule-bias) 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 5) (6 5) (7 5)>`. In other words, the probabilities of applying grammar rules 5, 6, and 7 to expand this non-terminal symbol are equal.

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.

The algorithms for implementing crossover and mutation are similar to those of GGP. However, the information maintained in the rule-biases list of different non-terminal symbols will be used to determine the crossover sites when crossover is performed. Similarly, mutation applies the information to decide which grammar rule should be used to expand a non-terminal symbol.

4.3 Adaptations of Extended Logic Grammars

From our experience in evolving the recursive even-n-parity program using GGP, we have observed that non-terminating programs with similar structures occur frequently in various generations. Consider the grammar depicted in Table 2,

```
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 L) T ],
                      s-expr(BOOL), [ ) ].
16:s-expr(BOOL)   ->  [ ( ], [ parity ],
                      s-expr(LIST), [ ) ].
17:s-expr(BOOL)   ->  [ ( ], [ first ],
                      s-expr(LIST), [ ) ].
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 (parity (rest L))))
```

and the program,

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

to be generated. The derivation trees of these two programs are shown in Figures 1 and 2 respectively. In the figures, the plain numbers are used to identify the sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in deducing the corresponding sub-trees.

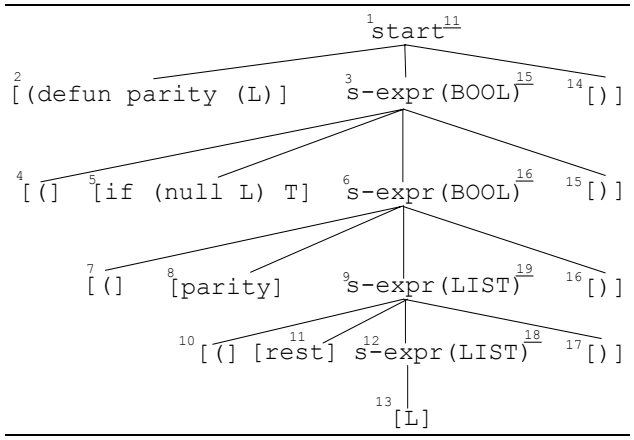

Figure 1: The derivation tree of the first program.

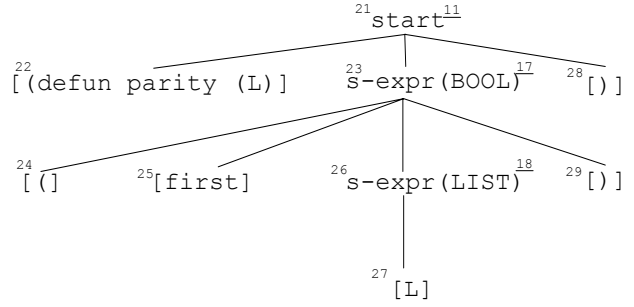

Figure 2: The derivation tree of the second program.

If the two programs are selected as parental programs to produce a new program through crossover. A non-terminating program,

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

will be created if the sub-trees 9 and 26 are exchanged. Since the grammar rule 18 is used to deduce the sub-tree 26 and the grammar rule 16 contains the non-terminal symbol `s-expr(LIST)` which is expanded to the sub-tree 9, 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. Therefore, adaptive GBGP retrieves the rule-biases list of the non-terminal symbol `s-expr(LIST)` of the grammar rule 16, and reduces the rule bias associated with the grammar rule 18. When modifying the rule bias, adaptive GBGP ensures that the rule bias will not be smaller than min-rule-bias.

5 Experiments

Two experiments have been performed repeatedly for 100 times to demonstrate the effectiveness of the adaptive mechanism described in Section 4.3. They differ in whether the adaptive mechanism is enabled or disabled.

In these experiments, the extended logic grammar in Table 2 is used by the systems to evolve recusive programs. The population size is 500 and the maximum number of generations is 50. The probabilities of performing crossover and mutation are respectively 0.6 and 0.3. The maximum depth of derivation trees generated in the initial population is 12. The maximum depth of trees produced by crossover and mutation is also 12. 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 50 is reached or a program that classifies all fitness cases correctly is found.

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, because 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.

In Section 5.1, we present the performance results of the system without the adaptive mechanism. In Section 5.2, the performance results of adaptive GBGP are described.

5.1 Non-adaptive GBGP

In the 100 trials, the system successfully evolves 46 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}. 44 of them can successfully solve all the problems. They are further analyzed manually and it is found that these 44 programs are correct recursive programs for the general even-n-parity problem.

The I(M, i, z) for z of 99% reaches a minimum value of 175,595 at generation 39 (Koza 1992). Since there are only 13 fitness cases, 175,595*13 = 2,282,735 fitness cases should be processed to find a general recursive program for the even-n-parity problem.

5.2 Adaptive GBGP

In the 100 trials, the system successfully evolves 49 programs that classify all fitness cases correctly. The generated programs are further analysized, 48 of them are correct recursive programs for the general even-n-parity problem. The curves in Figure 3 show the experimentally observed cumulative frequency of success F(M, i) of solving the problem by generation i using a population of M programs, where M is 500. It can be observed that adaptive GBGP performs better than non-adaptive GBGP.

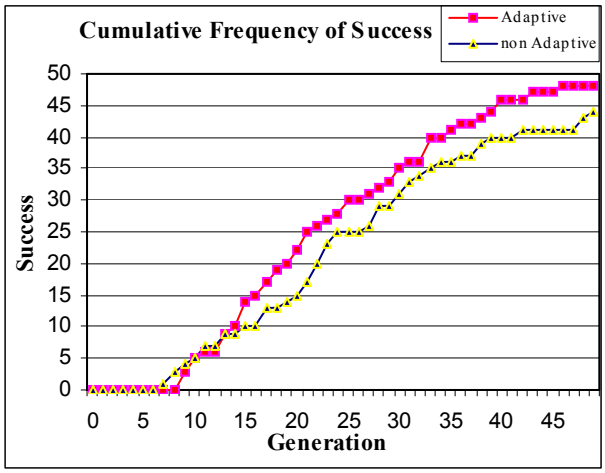

Figure 3: The performance curves showing cumulative frequency of success F(M, i) for the even-n-parity problem.

The curves in Figure 4 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 z is 0.99. The I(M, i, z) of adaptive GBGP reaches a minimum value of 151,092 at generation 40 (Koza 1992). This figure also shows that adaptive GBGP performs better than non-adaptive GBGP.

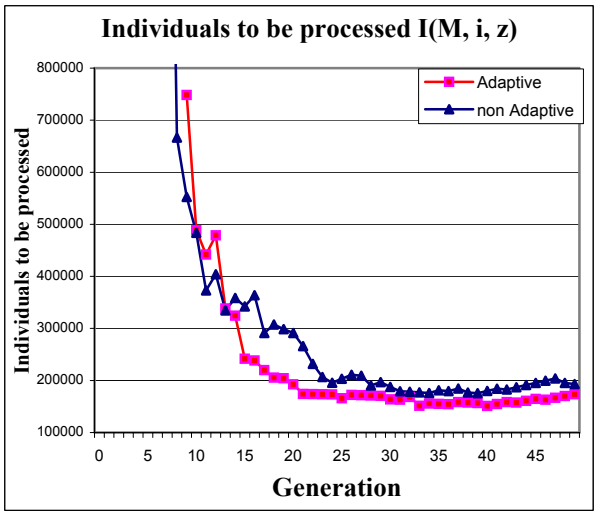

Figure 4: The performance curves showing I(M, i, z) for the even-n-parity problem.

Since there are only 13 fitness cases, 151,092*13 = 1,964,196 fitness cases should be processed to find a general recursive program for the even-n-parity problem. On the other hand, 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 94 times faster.

Table 3 summarizes the numbers of non-terminating programs generated by adaptive GBGP and non-adaptive GBGP in 100 trials. The average number of non-terminating programs generated by adaptive GBGP is 440.26, which is much smaller than that of non-adaptive GBGP (1,077.47). Moreover, the maximum number of non-terminating programs generated by adaptive GBGP is 1,173. This value is slightly higher than the average number of non-terminating programs generated by non-adaptive GBGP. In other words, adaptive GBGP generates much smaller number of non-terminating programs than non-adaptive GBGP in many trials.

	Adaptive GBGP	Non-adaptive GBGP
Average	440.26	1,077.47
Std.	193.00	557.27
Maximum	1,173	2,041
Minimum	167	237

Table 3: The numbers of non-terminating programs generated by adaptive GBGP and non-adaptive GBGP.

One of extended logic grammars modified by adaptive GBGP is shown in Table 4. It can be observed that the extended logic grammar can reduce the chance of generating non-terminating programs, and thus it accelerates the process of evolving recursive programs.

```
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 L) T ],
                          s-expr(BOOL), [ ) ].
16:s-expr(BOOL)       ->  [ ( ], [ parity ],
                          s-expr(LIST)<(18 1)
                          (19 5)>, [ ) ].
17:s-expr(BOOL)       ->  [ ( ], [ first ],
                          s-expr(LIST), [ ) ].
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 4: An extended logic grammar adapted by adaptive GBGP. The modified grammar rule is shaded.

6 Discussion and Future Work

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 1996). 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.

Similarly, our adaptive GBGP will not delete any existing rules. However, the probabilities of applying some rules will be reduced if they are inappropriate in certain 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 of the offspring created by performing crossover.

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 grmmars may allow adaptive GBGP to find the solution.

7 Conclusion

In this paper, we have proposed a technique to tackle the difficulties in learning recursive programs by dynamically modifying the grammar specifying the search space. The modified grammar reduce the chance of producing non-terminating programs, and thus it accelerates the process of evolving recursive programs. The technique is incorporated into an adaptive Grammar Based Genetic Programming system (adaptive GBGP). A number of experiments have been performed to demonstrate that the system can evolve recursive programs efficiently and effectively.

Acknowledgments

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

.Bibliography

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.

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

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.

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.

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. (1996*). Grammatical Bias for Evolutionary Learning*. Ph.D. Thesis. University of New South Wales.

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*.

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. (1999a). Polymorphism and Genetic Programming. In *Proceedings of the Fourth European Conference on Genetic Programming*.

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