Genetic Programming that Ensures Programs are Original

Shiu Yin Yuen and Shing Wa Leung

Abstract— **Conventional genetic programming (GP) does not guarantee no revisits, i.e., a program may be generated for fitness evaluations more than one time. This is clearly wasteful in applications that involve expensive and/or time consuming fitness evaluations. This paper proposes a new GP – non-revisiting genetic programming NrGP – that guarantees that all programs generated is original. The basic idea is to use memory to store all programs generated. To increase efficiency in indexing and storage, the memory is organized as an S-expression trie. Since the number of solutions generated is modest for applications involving expensive and/or time consuming fitness evaluations, the extra memory needed is manageable. GP and NrGP are compared using two GP bench mark problems, namely, the symbolic regression and the even N-parity problem. It is found that NrGP outperforms GP, significantly reducing the computational effort (CE) required. This clearly shows the power of the idea of ensuring no revisits. It is anticipated that the same non-revisiting idea can be applied to other types of GP to enhance their efficiency. A new CE measurement is also reported that removes some statistical biases associated with the conventional CE.**

I. Introduction

GENETIC programming (GP) [1] is one type of Evolutionary Algorithms (EA) that applies the idea of natural selection to computer programs. It is a relatively recent addition to the genetic algorithm, evolutionary strategies and evolutionary programming family. It can be considered as a natural variant to genetic algorithm (GA). Some researchers hold the view that GP is a generalization, rather than a special case of GA. The central idea of GP is to represent computer programs as chromosomes (individuals) within the GA framework. Different from conventional GA which has chromosomes with a fixed length, in GP, these chromosome programs are represented as tree structures composing of terminals and operators with dynamically changing shapes and sizes. To evolve, chromosomes with high fitness are chosen as parents to generate offspring by crossover and mutation. Since one is dealing with programs, crossover and mutation operators specific to GP are used. Under this favor-high-fitness selection method, offspring are supposed to be inheriting desirable characteristics from their parents and performing better than their parents. For example, two useful fragments of programs may be inherited and recombined to create a better program. However, by the same reasoning, parents with high fitness may be selected again and again, generating duplicated offspring, i.e. offspring programs that has been generated before, or non-original programs. Clearly, these duplicated offspring brings no contribution to the evolution but only consumes computational power in fitness evaluation. When solving problems whose fitness evaluations are expensive and/or time consuming, large amount of resources will be wasted in evaluating duplicated offspring. For example, in heating, ventilation and air conditioning (HVAC) engineering [2] and antenna design [3], it takes typically a few minutes to a day for one fitness evaluation. We shall use the term "the revisiting problem" to refer to the problem associated with resources being wasted in duplicated fitness evaluations.

In order to solve this problem, the non-revisiting GP, NrGP, is introduced in this paper. The non-revisiting idea serves as a natural add-on to the original GP. The essence of the idea is to use memory to store all generated solutions. To increase indexing and storage efficiency, the memory is organized as a S-expression trie. It completely eliminates the revisiting problem. Since the number of expensive and/or time consuming fitness evaluations is not high, the total memory required is manageable.

In the past, several researchers have attempted to incorporate memory in various ways into GP. In a sequel to [1], Koza [4] advocated using memory to save some branches in the individual tree as automatically defined functions (ADF) which can be re-used later. Bearpark and Keane [5] preserved some fragments of high fitness individuals in a pool and re-introduce them during the mutation phase. Teller [6] proposed using indexed memory to store and to reuse the result of sub-trees. He added the "Read" and "Write" functions into the terminal set of the GP system to access the indexed memory. The use of indexed memory worked as a pointer to connect one node to another in the individual tree. Recently, Walker and Miller [7] applied module encryption and decryption in Cartesian Genetic Programming to make it possible to evolve and reuse modules. The above researchers aimed to apply memory to store component parts in high fitness individual, making them immune to the disruptive effects of crossover and mutation. On the other hand, other researchers applied memory for other usages. Ok et al. [8] used memory to store terminals, by which unrelated terminal sets are identified and removed. Andre [9] applied memory into a multi-phased mapmaking program. He used memory to store the environment in the "mapmaking" phase. The stored environment was used to plan the actions and the movement of the robot in the "map-using" phase.

Shiu Yin Yuen is with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong SAR, China. E-mail: kelviny.ee@cityu.edu.hk

Shing Wa Leung is with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong SAR, China. E-mail: shinleung8@student.cityu.edu.hk

The work described in this paper was supported by a grant from CityU (7002304).

However, none of the above researches considered the revisiting problem. Moreover, stored component parts may increase the probability of revisits. The intended use of memory in this paper is entirely different to all of the above researches.

The original idea of NrGP comes from non-revisiting genetic algorithm (NrGA) [10, 11]. NrGA uses a binary partitioning tree memory to record all visited solutions. By saving and comparing individuals, no duplicated offspring will be generated. As a result, no fitness evaluation is wasted and no revisiting problem occurs. Significant performance improvement is reported. The idea is generic, for example, it can be applied to other types of search [12, 13, 14]. In each case, substantial performance improvement has been observed. Recently, a proof is obtained that the use of memory to assist an evolutionary algorithm can reduce the expected time complexity of finding the solution significantly for some problems [15].

This paper presents the general methodology to implement NrGP. Section II explains the idea and details of the NrGP. Section III gives the experiment settings. Results of experiments and discussions are shown in section IV. Finally, conclusions from experimental results and suggestions for the further works are presented in section V.

II. Non-revisiting Genetic Programming

A. *Idea of Non-Revisiting Genetic Programming*

The idea of Non-Revisiting Genetic Programming comes from the NrGA [10, 11], which use memory to record all evaluated individuals and evaluated results to increase the performance. Working as a generic extension part for a generic algorithm, the idea of "*non-revisiting stochastic search*" can be applied to many different algorithms. The empirical results in the researches of Yuen and Chow [10, 12, 13, 14] show significant improvements brought about by the idea of "*non-revisiting stochastic search*". To investigate the potential improvement that can be achieved by reducing the revisiting problem, in this paper, the idea of "*non-revisiting stochastic search*" is applied to the conventional GP.

Introduced by Koza [4], computational effort (CE) is commonly used to measure the performance of GP. CE is measuring the number of fitness evaluations that must be executed to yield a solution to a problem. By definition, the smaller is the CE, the better is the performance of the algorithm. As the revisiting problem is concerned with the wastage of computational resource on evaluating the fitness of the same individual repeatedly, it is clear that the CE performance will be decreased when some fitness evaluations are wasted.

By using NrGP, it is guaranteed that no redundant fitness evaluation occurs. Therefore, NrGP has a larger probability to get the optimal solution compared to conventional GP. Moreover, because visited points in the search space will never be revisited again, the search space becomes smaller after each fitness evaluation. As a result, in the long run, there exists an upper bound in the number of fitness evaluations while the conventional GP does not.

B. *Flow of NrGP:*

In this paper, conventional GP is used as the base to apply the "*non-revisiting stochastic search*" idea. Compared to conventional GP, only two simple sub-processes, i.e., *saving* and *revisit checking*, are inserted into the original algorithm. The *saving* process records the visited points (evaluated individuals) while the *revisit checking* process checks whether the new individual is revisited or not. They will be called when new individual is produced. If the new individual is determined as having been revisited, the reproducing operation will be called again to generate a new individual until it is not a revisit. In this way, it makes sure that all programs generated are original.

Let *M* be the population size, the following code constructs NrGP:

```
FOR ( i := 1 to M)
{
   Generate individual i;
   WHILE (individual i revisited)   Generate individual i;
   Save  individual i ;
}
WHILE ( Termination condition  NOT reached)
{
   FOR ( i := 1 to M)  Evaluate Fitness of individual i;
   i := 1;
   WHILE ( i < M)
   {
      Select Parents;
      IF( crossover operation is selected)
      {
         DO
         {
            Generate offspring i by crossover;
         } WHILE (Offspring i revisited);
      }
      ELSE
      {
         DO
         {
            Generate offspring i by mutation;
         } WHILE (Offspring i revisited);
      }
      Save Offspring i;
      i := i+1;
   }
   IF (best offspring better than historical best) THEN
      update historical best;
   FOR ( i := 1 to M)  individual i := Offspring i;
}
```

Code in bold is the extension added to the conventional GP. Compared to the original algorithm, only a few steps are added into the system. Note that though the while loops involve repeated solution generation, it actually involves insignificant computational overhead in applications involving expensive and/or time consuming fitness evaluations.

C. *Implementation of NrGP*

As mentioned before, the main idea of NrGP is to store and compare individuals in all generations. By comparing the new individual with the evaluated individuals, the revisiting problem can be prevented. As individuals in GP are generally represented as variable structure binary tree, the problem of how to store and compare individuals effectively needs to be solved. In this paper, we suggest a fast and effective approach for storing and comparing binary trees with different structures and sizes. Individuals in a GP are programs, which are in turn variable structure binary trees of terminals and operators. Such a binary tree is first translated to a variable length string. The most common way to translate the binary tree is the S-expression as shown in fig 1. It converts the binary tree to a string of terminals and operators. For efficient query of whether an S-expression is a revisit, each S-expression is simply stored as a path in the trie [16], as shown in Fig.2. Each node in the trie has *K = (Size of Function set + Size of Terminal set+1)* child node pointers. Each non-root node in the trie represents a terminal or an operator in the S-expression string. The additional sign *EoP* represents the end of program. A new individual is saved as a path from the root to an *EoP* by creating the necessary nodes, as follows: elements in the strings are used to indicate the path. At each node of the tree, the corresponding child node pointer will be checked. If the corresponding child node pointer is pointing to the desired node, the node will be chosen. Otherwise, a new node will be created to the corresponding child node pointer. As each path in the tree is representing an individual, to check whether a string is a revisit simply entails checking whether the corresponding path ending in *EoP* already exists in the trie.

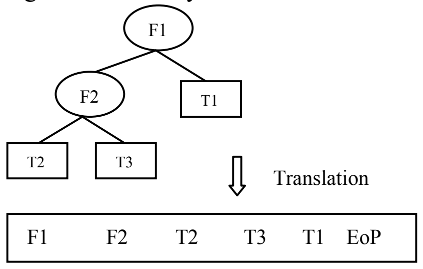

Fig. 1. An example of translating a binary tree to a node string. Using S-expression notation, nodes are read in order: current node, left child node and then right child node to convert recursively. The first chosen node is the root node of the individual tree. Therefore, F1 is the first element in the translated string. After that, the next node is the left child node of the current node, so the current node is changed to node F2. Similar to the previous operation, the second element in the translated string is F2. Then, change the current node to leaf child node T2. Since node T2 has no child node, return to node F2 and choose right child node T3 and so on. Finally, when the translation finishes, an "EoP" sign is appended to the translated string.

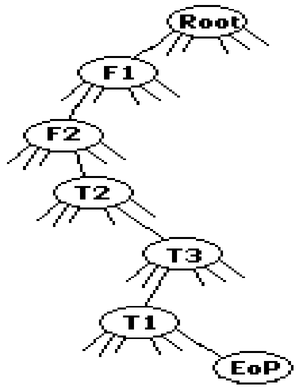

Fig. 2. An example of the trie used in this paper. Assume that the size of the function set is 2, i.e.,{F1,F2}, and the size of the terminal set is 3, i.e., {T1,T2,T3} such that each node in the tree has 2+3+1 = 6 child node pointers. In the beginning, there is only root node in the tree. For saving the individual shown in Fig.1, a path is constructed. Lines without node are null child node pointers.

1) *Population Initialization*

Individuals in GP are represented as binary trees. Two methods commonly used to build a tree are *grow* and *full* method. The difference between the *grow* and the *full* method is that the *full* method generates trees of which all the leaf nodes are in the same depth. For example, the tree shown in Fig.1 is not a full tree. The size and the structure of the full tree are rather limited. Since the first generation should have more variations in sizes and structures, all trees are generated by the *grow* method which constructs trees with more varied sizes and shapes.

To generate trees by the grow method, let F_{size} and T_{size} be the size of function set and terminal set respectively. Each node has probability $\left(\frac{F_{size}}{F_{size}+T_{size}}\right)$ and $\left(\frac{T_{size}}{F_{size}+T_{size}}\right)$ to be a function node and terminal node respectively. If the node is chosen as a function node, it will have two child nodes. Otherwise, the node will become a leaf node.

2) *Crossover and mutation operator*

Crossover and mutation are the main operators to produce new offspring in GP. The crossover operation exchanges some genes between two parents and the mutation operation makes some changes to the genes of the parent to produce offspring. As individuals in GP are represented in a tree structure, genetic information of each individual are represented as sub-trees.

In the crossover and mutation operation, a node will be chosen randomly as the cutting point. To perform crossover, the sub-trees under this node will be swapped between parents to generate offspring. To perform mutation, the sub-tree under this node will be deleted and a new sub-tree is regenerated by the *grow* method.

To prevent bloating [18], an operator similar to size-fair crossover operator is used [17]. The first cutting point of crossover is selected randomly in the first parent. The position of the first point is used as a constraint to make sure that the sub-tree under the second selected point will not cause over size.

To select a cutting point, two parameters are generated randomly: *depth* and *path*. The *depth* parameter is used to indicate the maximum depth of the selected cutting point. The range of the *depth* parameter is from 1 to the user defined maximum depth. Starting from the root node, the parameter *path* is generated randomly at each node. The pointer moves to the left or right child node of the current node if *path* is an even or odd number respectively. The cutting point will be selected when the pointer is pointing to the leaf node or the depth of the current node reaches *depth*.

After selecting the first cutting point from the first parent, the second cutting point will be selected from the second parent. The second cutting point is selected restrictively so that the sum of depth of the first cutting point and the sub-tree under the second cutting point is less than or equal to maximum allowed depth.

III. Experiment

Symbolic regression and even *N*-parity problem are used as the testing problems for comparing the performance of NrGP with conventional GP. 100 independent trials are conducted and the t-test is used to ensure that results are statistically significant. The initial tree depth is set to five levels and the maximum tree depth of an individual is limited to fifteen levels. A PC with 3.0GHz CPU and 1GB memory is used. The programming language used is C++.

A. *Symbolic regression*

Symbolic regression is a widely used benchmark suggested by Koza [1, 4]. The problem is to find an equation $y = f(x)$ that fits the given set of data. The equation will be a combination of functions in the function set {addition, subtraction, multiplication and protected division (return 1 if divided by zero)} and terminals in the terminal set {X, random constants}. The given data is a set of points on the x-y coordinates (e.g.: $d_1 = (x_1,y_1)$, $d_2 = (x_2,y_2)$, ...). To determine the accuracy of the equation, the x-coordinate value of each point will be substituted into the equation, and then the output of the equation will be compared with the given y-coordinate value of the point to calculate the error.

As the aim of symbolic regression is to determine the equation which fits all the given points, the sum of the absolute errors between the y-coordinates y_n of the given point $d_n = (x_n, y_n)$ and the output g_n of the equation $g_n = f(x_n)$ will be used as the fitness function: i.e., $fitness = \sum |y_n - g_n|$.

In this paper, three polynomials with different orders were used in the experiment. They are

1) Quartic function $f(x) = x^4 + x^3 + x^2 + x$,

2) Quintic function $f(x) = x^5 - 2x^3 + x$ and

3) Sextic function $f(x) = x^6 - 2x^4 + x$.

The parameter settings are shown in Table I. As the size of the terminal set {x, random constant} is undeterminable, the tree used to store individuals cannot be constructed. To solve this problem, the random constant is not included in the terminal set in this experiment.

TABLE I
THE PARAMETER SETTINGS USED IN SYMBOLIC REGRESSION

Population size:	100
Generation count:	100
Selection Method:	Tournament with size 5
Tree Generation:	Grow
Initial Tree depth:	5
Maximum Tree depth:	15
Crossover probability:	0.9
Mutation probability:	0.1
Terminal set:	X
Function set:	+,-,*,%(protected division)
Fitness cases:	50 points, using evenly spaced values of x between -1 & 1
Hit:	Error <0.001

B. *Even N-parity problem*

The even *N*-parity problem is another common benchmark problem in GP research [1].The even N-parity function returns *TRUE* if the number of *TRUE* in all *N* inputs is even, otherwise, it returns *FALSE*. In practical applications, the even *N*-parity function is often used as an error checking code that checks the accuracy of data storage and data transmission. Similar to the symbolic regression problem, the aim of the even *N*-parity problem is to find the Boolean equation (e.g. *output = (X1 AND X2) OR X3*) which outputs the same result as the even *N*-parity function. To solve the even *N*-parity problem by GP, primitive Boolean operators {ADD, OR, NAND, NOR} are used as the function set, and the *N*- bits input are used as the terminal set.

The performance of NrGP is compared with that of conventional GP in even 3-parity problem and even 4-parity problem. Table II shows the settings of this experiment.

TABLE II
THE PARAMETER SETTINGS USED IN EVEN – N PARITY PROBLEM

Population size:	400
Generation count:	50
Selection Method:	Tournament
Selection Method:	Tournament with size 5
Tree Generation:	Grow
Initial Tree depth:	5
Crossover probability:	0.9
Mutation probability:	0.1
Terminal set:	bit 0 to bit N-1
Function set:	AND,OR, NAND,NOR
Fitness cases:	All 2^N combinations of the N Boolean arguments
Hit:	Error =0

C. *Fitness measurement*

1) *Computational effort*

CE is introduced by Koza [1] to measure the effort required to solve the problem. It is calculated by the equations shown in eqn (1). It includes four steps. Firstly, let *M* be the population size. *NT* independent trials are made and the number of successful trials *N(M,i)* is recorded. *P(M, i)*, the probability of success at generation *i* is calculated as *N(M,i)* divided by *NT*. Secondly, the required number of independent trials *R(i,z)* to solve the problem with a user defined high probability *z* (e.g. *z=0.99*) is calculated. Thirdly, the total number of individuals *I(M,i,z)* produced in *R(i,z)* independent trials is calculated. Finally, *CE* is determined to be the minimal number of individuals *I(M,i,z)* as *i* is allowed to vary. To explain briefly, the physical meaning of *CE* is the minimal number of individuals that has to be produced to get the solution in high probability. As the total number of individuals is equal to the number of fitness evaluations, *CE* can also be defined as the minimal number of fitness evaluations to yield the solution in high probability, ignoring solution generation cost. As *CE* is measuring the number of the fitness evaluations but not the actual time consumed in the whole experiment, it is a fair measure for computers with different processing speeds.

$$\begin{aligned} P(M,i) &= \frac{N(M,i)}{NT} \\ R(i,z) &= ceil\left(\frac{\log(1-z)}{\log(1-P(M,i))}\right) \\ I(M,i,z) &= M \times R(i,z) \times (i+1) \\ CE &= Min_i I(M,i,z) \end{aligned} \qquad (1)$$

2) *A New Computational Effort Measure*

In *CE*, we calculate the cumulative probability of success in generation *i* through all independent trials. However, the calculation of *P(M,i)* is not so accurate. If the number of independent trials is too small, the result does not have a high confidence level. For example, success once in two trials of course does not mean that it would be successful fifty times in one hundred trials. Moreover, Christensen and Oppacher [19] reported that the *ceil* (round up) operation in the calculation underestimated the true CE. In the literature, researchers use different number of independent trials *NT* in their applications arbitrarily, partly dependent on the time needed to run the experiments. In a recent research, Niehaus and Banzhaf [20] found that the number of independent trials used affected the degree of underestimation.

To make the calculation of *CE* more statistically accurate and unaffected by underestimation, we suggest to use the method proposed by Yuen et al. [21]. It is a method to estimate the probability *P(M, i)* with high confidence.

Note that the above fulfils the three characteristics of binomial experiments. First, involves repeated number of NT trials. Second, there are only two possible outcomes: success (Hit) or failure (Not Hit). Third, all the trials in the experiment are independent.

Let *x* be the number of successes in generation *i* after NT number of independent trials. Eqn (2) denotes the binomial distribution.

$$b(x; NT, P_{est}(M,i)) = \binom{NT}{x} P_{est}^{x}(M,i) \times (1-P_{est}(M,i))^{NT-x}$$
$$x = 0,1,\ldots,NT \qquad (2)$$

Although strictly speaking, it is an estimation problem, it is somewhat easier to explain using a hypothesis testing framework:

$$\begin{aligned} H_0&: \quad P = P_{est} \\ H_1&: \quad P > P_{est} \end{aligned} \qquad (3)$$

Where H_0 is the null hypothesis and H_1 is the alternative hypothesis.

Let *X* be the statistic, which is the number of successes in *NT* trials. Then $X \le x$ is the acceptance region and $X \ge x+1$ is the critical region. Thus the probability of type 1 error, which is the level of confidence α, can be calculated by the following equation:

$$\alpha = P(Type\,I \quad error) = P(X \ge x+1 \mid P = P_{est})$$
$$\alpha = \sum_{x=N(M,i)+1}^{NT} b(x; NT, P_{est}(M,i)) \qquad (4)$$

Given *NT*, *N(M, i)* and a user defined level of confidence α, for P_{est} much lower than *N(M, i)/NT*, the null hypothesis will be rejected. We choose the value of P_{est} as the maximum value such that H_0 is rejected. Let this value be P_{lcf}. In statistics, this value is the *lower confidence bound* of the probability of success, with *confidence level* α [22]. Then we replace *P(M,i) by* P_{lcf} *(M, i)* in eqn (1), i.e.

$$\begin{aligned} \alpha &= \sum_{x=N(M,i)+1}^{NT} \frac{NT!}{x!(NT-x)!} p^{x}{}_{est}(M,i) \times (1-p_{est}(M,i))^{NT-x} \\ R(i,z) &= ceil\left(\frac{\log(1-z)}{\log(1-p_{\mathrm{lcf}}(M,i))}\right) \\ I(M,i,z) &= M \times R(i,z) \times (i+1) \\ CE &= Min_i I(M,i,z) \end{aligned} \qquad \mathbf{(5)}$$

The advantage of this new CE is that the lower confidence bound normalizes the statistical uncertainty associated with using different number of independent trials *NT* observed in [20]. Moreover, the *ceil* operation no longer underestimate the true CE as reported in [19], as the operation now is exactly calculating the (integer) number of repeated trials in a probabilistic amplification procedure to achieve the desired confidence level.

Because of the above advantages, we suggests that in the future, CE should be written as CE (% confidence). For example, CE (99.9% confidence) means a confidence level of 99.9% or an α of 0.1%.

IV. Results

A. *Results of symbolic regression*

Table III shows the result of the symbolic regression experiment on Quartic function. The performance of NrGP is significantly better than that of conventional GP. Using NrGP, the hit rate is increased by 19%, the computational effort is reduced by about 42% and the average error is reduced by about 77%. It illustrates that NrGP gives a large improvement in this simple symbolic regression problem. It can be concluded that NrGP improves the performance by reducing the revisiting problem; the large reduction in the *CE* means that the revisiting problem in this problem is very serious.

Table IV shows the result of the experiment on Quintic function. By comparing the results between GP and NrGP in hit rate, average error and computational effort, the improvement of NrGP is also significant. The hit rate is increased by about 18%, the computation effort is reduced by about 40%, and the average error is reduced by 51%.

TABLE III
RESULTS OF SYMBOLIC REGRESSION IN SOLVING QUARTIC FUNCITON

Function:	Quartic function $f(x)=x^4+x^3+x^2+x$	
Method:	GP	NrGP
Average Fitness: (Error)	0.89213706	0.20253827
Standard Deviation:	1.75437996	0.62706843
Average Generation used:	42.83	24.75
Hit rate:	69%	88%
Computational Effort:	11200	6500
Computational Effort : (99.9% confidence)	18200	9800
t value: (t- Test)	3.7014 (99.97%)	

TABLE IV
RESULTS OF SYMBOLIC REGRESSION IN SOLVING QUINTIC FUNCTION

Function:	Quintic function $f(x)=x^5-2x^3+x$	
Method:	GP	NrGP
Average Fitness: (Error)	0.97472161	0.48003359
Standard Deviation:	1.39990242	0.65154054
Average Generation used:	78.36	66.17
Hit rate:	32%	50%
Computational Effort:	60000	36000
Computational Effort : (99.9% confidence)	126000	66300
t value (t- Test):	3.2037(99.84%)	

Table V shows the result of the experiment on Sextic function. Though NrGP makes no improvement in the hit rate, it reduces by 35.5% in computational effort and 41% in average error.

The t-tests on fitness show that the fitness improvement due to NrGP is statistically significant.

TABLE V
RESULTS OF SYMBOLIC REGRESSION IN SOLVING SEXTIC FUNCTION

Function:	Sextic function $f(x)=x^6-2x^4+x$	
Method:	GP	NrGP
Average Fitness: (Error)	1.01978341	0.60442356
Standard Deviation:	1.279339336	0.72006868
Average Generation used:	81.59	78.31
Hit rate:	31%	31%
Computational Effort:	75000	48400
Computational Effort : (99.9% confidence)	178600	103400
t value :(t- Test)	2.8292(99.49%)	

The results show that the improvement in the simpler problem (Quartic function) is more significant than that in the complex problem (Sextic function). The decrease in the improvement is the most significant in hit rates. It drops from about 19% in Quartic function to 18% in Quintic function and then to no improvement in Sextic function. However, it does not mean that NrGP has no improvement in more complex problems, the decrease in computation effort and average errors are still over 35% and 40% respectively in the Sextic function.

B. *Results of even N- parity problem*

Table VI shows the result of solving the even 3-parity problem and the even 4-parity problem. It illustrates that both conventional GP and NrGP can solve the even 3-parity problem easily. Both conventional GP and NrGP find the solution with very high probability (hit rate > 90%). Using NrGP, the computational effort is reduced by about 58%.

In the even 4-parity problem, the result demonstrates that the performance of NrGP is significantly better than GP. Not only is the hit rate increased by 26%, but also the average error and the computational effort are decreased by about 52% and 63% respectively. The result shows that the performance of NrGP is much better than that of conventional GP. The t-tests on fitness show that the fitness improvement due to NrGP is statistically significant.

TABLE VI
RESULTS OF EVEN N-PARITY PROBLEMS

Function:	Even 3-Parity Problem		Even 4-Parity Problem	
Method:	GP	NrGP	GP	NrGP
Average Fitness: (Error)	0.07	0	2.9	1.38
Standard Deviation:	0.26	0	2.38	1.57
Average Generation used:	15.87	8.91	48.47	42.56
Hit rate:	93%	100%	21%	47%
Computational Effort:	23200	9600	400000	147200
Computational Effort: (99.9% confidence)	34800	14400	800000	239200
t value :(t- Test)	2.7298 (99.31%)		5.3187(99.99%)	

In summary, the results in the even 3-parity problem and the even 4-parity problem once again show that the performance of the classic GP can be improved by alleviating the revisiting problem. In particular, NrGP performs better when the revisiting problem in GP is more serious in the application scenario. For the even parity problems, the performance of NrGP is better when the problem is more complex.

V. Conclusion and further work

In this paper, we propose an extension to genetic programming (GP), called non-revisiting genetic programming (NrGP), which applies memory to record all visited points in the search space. By recording the visited points, NrGP guarantees that no point will be revisited in the search space. Therefore no computational resource will be wasted in duplicated fitness evaluation. Our contention is that in expensive and/or time consuming fitness evaluations, duplicated fitness evaluations is an unjustifiable waste of resources and should be avoided, and for such applications, the memory required to store all visited solutions are manageable (since the number of fitness evaluations will not be large). To further increase solution generation and memory storage efficiency, an S-expression trie data structure is also reported for storing visited points and querying whether a search point is a revisit.

The performance of conventional GP and NrGP is compared by using two bench mark GP problems, namely, symbolic regression problem and even N-parity problem. In each case, NrGP brings significant performance gains to GP.

From the empirical results, we conclude that the idea of "*non-revisiting stochastic search*" is a good extension part to conventional GP that delivers a performance gain. Moreover, the idea can be applied to other more advanced types of GP and performance gain is expected. This is one area for future work. Another area is to investigate techniques that use the memorized data more effectively. This includes adaptive mutation and other search-history-driven techniques.

An independent contribution of this paper is the introduction of a new computational effort (CE) measure. We propose to eliminate the estimation uncertainty inherent with sample sizes in the original CE by computing the statistical lower confidence bound. We suggest using the notation CE (% confidence) to denote this new CE measure. Since the number of independent trials for GP and NrGP are deliberately set to be the same in our experiment, the new CE does not make any qualitative difference to our results in the paper. However, we believe that the new CE will be useful for an unbiased comparison of GP in the literature. Since CE is a generic, algorithm independent measure, it can equally be applied to compare other evolutionary algorithms as well.

References

[1] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[2] K. F. Fong, V. I. Hanby and T. T. Chow, "HVAC system optimization for energy management by evolutionary programming," *Energy and Buildings,* vol. 38, pp. 220-231, 2006.

[3] Q. X. Chu, K. F. Chan and C. H. Chan, "Parallel FDTD analysis of active integrated antenna array," *Microwave and Optical Technology Letters,* vol. 34, pp. 317-319, 2002.

[4] J. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, 1994.

[5] K. Bearpark and A. J. Keane, "The use of collective memory in genetic programming," in *Knowledge Incorporation in Evolutionary Computation* Y. Jin, Ed., pp. 15-36, 2004.

[6] A. Teller, "Genetic programming, indexed memory, the halting problem, and other curiosities," in *Proceedings of the 7th Annual Florida Artificial Intelligence Research Symposium (FLAIRS -94),* 1994, pp. 270-274.

[7] J. A. Walker and J. F. Miller, "The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming," *IEEE Transactions on Evolutionary Computation,* vol. 12, 2008, pp. 397-417.

[8] S. Ok, K. Miyashita and S. Nishihara, "Improving performance of GP by adaptive terminal selection," *PRICAI 2000 Topics in Artificial Intelligence: 6th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2000),* pp. 435-445, 2000.

[9] D. Andre, "Evolution of mapmaking: learning, planning, and memory using genetic programming," in *Proceedings of IEEE Conference on Evolutionary Computation,* 1994, pp. 250-255.

[10] S. Y. Yuen and C. K. Chow, "A non-revisiting Genetic Algorithm," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2007)* , 2007, pp. 4583-4590.

[11] S. Y. Yuen and C. K. Chow, "A Genetic Algorithm that Adaptively Mutates and Never Revisits," *IEEE Transactions on Evolutionary Computation,* to be published.

[12] S. Y. Yuen and C. K. Chow, "Applying non-revisiting genetic algorithm to traveling salesman problem," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2008)* , 2008, pp. 2217-2224.

[13] S. Y. Yuen and C. K. Chow, "A non-revisiting simulated annealing algorithm," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2008)* , 2008, pp. 1886-1892.

[14] C. K. Chow and S. Y. Yuen, "A non-revisiting particle swarm optimization," in *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2008)* , 2008, pp. 1879-1885.

[15] C. W. Sung and S. Y. Yuen, "On the analysis of the (1+1) evolutionary algorithm with short-term memory," in *Proceedings of IEEE Congress Evolutionary Computation (CEC 2008)* , 2008, pp. 235-241.

[16] E. Fredkin, "Trie memory," in *Communications of the ACM*, 1960, vol 3, pp. 490-499.

[17] R. Crawford-Marks and L. Spector, "Size control via size fair genetic operators in the PushGP genetic programming system," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)* , 2002, pp. 733-739.

[18] A.P. Engelbrecht, Computational Intelligence, 2ndEd, Wiley 2007

[19] S. Christensen and F. Oppacher, "An analysis of Koza's computational effort statistic for genetic programming," *Genetic Prgramming: 5th European Conference (EuroGP 2002)* , 2002, vol. 2278, Lecture Notes in Computer Science, pp. 182–191.

[20] J. Niehaus and W. Banzhaf, "More on computational effort statistics for genetic programming," *Genetic Programming: 6th European Conference (EuroGP 2003)* , 2003, vol. 2610, Lecture Notes in Computer Science, pp. 164–172.

[21] S. Y. Yuen, C. K. Fong and H. S. Lam, "Guaranteeing the probability of success using repeated runs of genetic algorithm," *Image and Vision Computing,* vol. 19, pp. 551-560, 2001.

[22] E.L. Lehmann, *Testing statistical hypotheses*, 2nd Ed., Wiley, 1986.