A Stepwise Evolution of Functions

NOR ZAINAH SIAU, CHRISTOPHER J. HINDE, ROGER G. STONE
Department of Computer Science
Loughborough University
Loughborough
UNITED KINGDOM
{n.z.siau, c.j.hinde, r.g.stone} @lboro.ac.uk

Abstract: - A Genotype-Phenotype mapping in most Genetic Programming (GP) systems uses a predefined and rigid grammar definition. This method has been successful in producing the required solution. However, it can only be used to solve a limited set of problems. In this paper, a Teachable GP (TGP) system is proposed. An external GP system evolves a complete computer program, which acceptable solution is then added automatically to the existing grammar definition as a function and made available to the TGP system. This dynamic grammar definition allows for a more complex program to be generated, solving more complex problems. Experiments are performed to compare performances between GP without the added function, GP with a user-defined function and GP with the evolved function and results shows that GP with an evolved function is comparable to the GP with user-defined function and outperformed GP without function.

Key-Words: - Incremental XML Grammar Definition, Teachable Genetic Programming System, Function Evolution.

1 Introduction

There have been many evolution techniques introduced since the work of Cramer [3], who explicitly uses Genetics to generate computer programs. John Koza [6] later popularised this area and called it Genetic Programming (GP). The Koza-styled GP is based on a tree representation of a computer program in LISP.

In recent years, different variants of GP have emerged following Koza's basic idea, especially the separation of genotype and phenotype, proposed by Banzhaf [2]. Many new issues have been encountered since then, such as characteristics inheritance, generating syntactically correct program and producing a complete program. We tackle these issues by applying a full-syntax XML grammar definition [10]. The grammar definition provides rules to assist the translation of genetic codes (genome) into particular computer program syntax (phenome). As a result, this avoids the need to implement a repair procedure.

Koza later introduces a modular solution into the standard GP called Automatically Defined Functions (ADF) to scale to a larger and more complex program solution [7].

The main contribution of this paper is the teaching of a GP system to solve various user-defined problems by incorporating an evolved function into its grammar. In particular, we evolve a 'swap' program, external to a 'sorting' program. This speeds up the performance of the GP system by reducing both the number of generations required and the time taken to achieve a fit solution. It is also closer to the way in which a human might be taught to program, by incrementally increasing their library of program fragments. Our challenge is to get the right function to be included in the evolutionary system without introducing unnecessary mechanisms within the program.

This paper marks our initial work, which is a part of a larger project to produce a teachable web information extraction (TWIE) system. Our TWIE aims to discover specific pieces of information on the Web by evolving regular expressions automatically, with some assistance from a human. The regular expression notations are defined as productions in XML to match the DOM tree structure and the data pattern of the information.

The remainder of this paper is arranged as follows: Section 2 describes the related work before providing information on our approach in the following sections. The process of mapping the genotype to the phenotype is presented in section 5. This is then followed by details of the experiments and the results in section 6, and finally conclusions are drawn, including the future direction of this work.

2 Modularised solutions techniques

In this section, we briefly review the most relevant work on function evolution where an improvement in performance has been observed compared to the standard GP.

John Koza expanded his earlier work to apply program decomposition or modularisation. This allows for automatic creation of parameterized functions that can be invoked from the main program while the GP is concurrently being evolved. GP with ADF automatically breaks down a program into a set of modularised subprograms during runtime, with each solving a sub problem with the capability for reuse and then reassembling to solve the original overall problem. This approach allows for generation of a larger and more complex program and has a benefit of significant reduction in the computational effort compared to GP without ADF. However, before an evolution commences, the number of functions and their parameters need to be defined, although during runtime, these parameters are allowed to change with no human intervention. More details of ADF can be found in [7].

Angeline and Pollack [1] introduce Genetic Library Builder where a randomly selected parameterised subroutine can be compressed by an operator and placed in this library. How viable a subroutine is depends on the frequency that it is called. Before this compressed subroutine can be called, it will be expanded using another operator to its original definition. This way, the subroutines are protected. Any unused subroutines in the library will be removed.

Another variation of ADF can be seen in the work of Harper and Blair [4]. Using Dynamically Defined Functions (DDF), functions are dynamically created using a core grammar represented in BNF notation and these functions are then automatically appended to this grammar. Contrasting ADF, DDF does not require the user to specify the number of functions and their parameters prior to evolution. The functions, which may have any number of parameters, can be invoked by the main program, independent of any special-purpose operators or constraints.

Because there is no specific size of individuals being set, this method is facing a danger of insufficient integers to complete a code as well as program "bloat". Individual is either discarded or a specific method is then put in place to overcome these issues. The functions in DDF are not evaluated and are not defined to do a specific task.

A more recent work, which introduces Cartesian GP (CGP), is by Miller and Harding [9]. CGP, originally developed by Miller and Thomson [8] is concerned with providing an effective method for evolving digital electronic circuits. In CGP a computer program is encoded in the form of a linear string of integers representing an indexed graph. Like the standard GP, CGP also applies genotype-phenotype mapping. The genome represents some functions and node connections, producing an executable program. Although the genomes are fixed length, the phenomes length varies. This is because the nodes, which are encoded by a number of integers, are not required to be connected to each other. Any unconnected nodes will not be processed and do not have any effect to the program's behaviour.

All the above works are based on top-down approach, where functions are derived from the main program. In the context of our work, we use the bottom-up approach where we require useful function to be evolved separately from the main and new function definitions are added to the core grammar for future use.

3 Teachable GP System

In this section, our Teachable Genetic Programming (TGP) system is described. TGP system aims to automatically evolve an independent program, build it as a function so it can be used or reused by the main program to solve more complex problems.

Our TGP system generates a PERL program and so do the productions defined in the XML grammar. The grammar definition is described further in section 4.

3.1 Basic Structure

Figure 1 shows the basic structure of our TGP system. TGP system evolves programs that can reuse previously evolved programs as subroutines. Once this subroutine is completed, it is automatically added into the core grammar, ready to be called by TGP system.

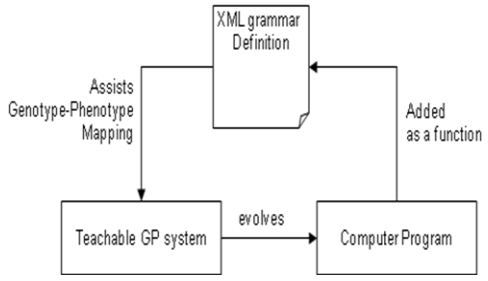

Fig. 1: TGP system structure.

The TGP system uses a fixed block length genotype and a 'clean' grammar definition, introduced in our previous research [10] that aims to remove dependency on an error correcting mechanism to produce an error-free and complete program. The grammar for the language is defined in XML format, separate from the system.

3.2 Function Structure

Figure 2 shows the general grammar definition of a function. The function evolution is triggered if a gene maps to a <function> and it would lead to several kinds of function calls including:

swap()	-	function with no parameter
swap(wvar)	-	function with a single parameter.
swap(wvar, wvar)	-	function with 2 parameters. Parameters are not limited to 2.
(wvar) = swap(wvar)	-	function with a single parameter and return a single value.
(wvar, wvar) = swap(wvar, wvar)	-	function with multi parameters and expecting a return of multi values.

But because of these many possibilities, the function evolution would require a longer time to get to the right solution. Therefore, being the purpose of our initial experiment is to teach the GP system to evolve a function, we decided to simplify the function generation (Figure 2) by constraining the way it is structured as represented in Figure 3.

```
<function>   ::=   <fcall>                    |
                    <retf><fcall>
<fcall>      ::=   <funcID> "(" ")"            |
                    <funcID>"("<params>")"
<funcID>     ::=   "swap"
<params>     ::=   <param>                    |
                    <param> "," <params>
<param>      ::=   <var>
<retf>       ::=   <param> "="                |
                    "(" <params> ")" "="
```

Fig. 2: General function grammar definition in BNF.

```
<function>          ::=   <retf><fCall>
<retf>              ::=   "(" <var> "," <var> ")" "="
<fCall>             ::=   "swap"  "("  "N1"  ","  "N2"
                          ")"
```

Fig.3: Reduced functions grammar definition in BNF.

N_i as used here is a special notation representing a pseudo non-terminal that is recognised by the program. On the right hand side of the production, all non-terminals are numbered from 1 so that N1 means a repeat of the first non-terminal and N2 repeats the second and so on.

3.3 Fitness Functions

The fitness function determines the direction of the evolved program and provides a greater impact on the success of the evolved program. Fitness is calculated to determine how close the output produced by a particular phenome is to the expected output [6]. Because the formal specification of a sort [5] is time consuming as it tests all possible pairs of integers, we use a simpler version by Withall [11]. With this fitness function, the evaluation is based on comparison of adjacent elements in the list rather than all element pairs and every conjunctive goal will contribute to the fitness score. It is important to note that the swap program has a different objective from the sorting program, therefore a new fitness function needs to be designed. In contrast to the traditional fitness function, the fitness evaluation used here is derived from the formal specification of the desired function. We have had successful experiments using the formal specifications to define complete and concise fitness functions, outperformed a simple input/output pair. Figure 4 shows both fitness functions used in the experiment where @L is the original list and @N is the result list.

```
Sorting Program:
$fitness++ if(bageq(\@L, \@N));
if($#N > 0){
 for my $x (0..$#N-1){
    $fitness++ if($N[$x]<=$N[($x+1)]);
 }
}

Swap Program:
$fitness++ if(bageq(\@L, \@N));
if($#N > 0){
 $fitness++ if($N[$x]==$L[($x+1)]);
 $fitness++ if($N[$x+1] == $L[($x)]);
}
```

Fig. 4: Fitness functions for sort and swap program.

3.4 Breeding Strategy

The selection strategy used for choosing individuals for reproduction is the Roulette Wheel Selection. Individuals are selected proportionate to their fitness value, leaning towards the fitter one.

Finally, new individuals are created with the aid of uniform crossover and mutation. 50% crossover probability is chosen where each gene of the parents has the potential of being swapped. Mutation is set to affect 1 in 10 genes and can take place anywhere in the genome. The point of mutation (gene) will be replaced with a random integer between 1 and 255.

4 Incremental Grammar

The XML grammar definition specifies the syntax (presence and order of elements), which conforms to a certain rule. Our core grammar definition consists of a list of major generic kinds of program statements arranged in a hierarchical form, and may have some sub elements, which define them. Figure 5 shows the structure of these components and their subcomponents. This grammar is designed as a "well formed" and "valid" XML document, validated against a Document Type Definition (DTD). The grammar definition could be represented in another language, with a small modification in the GP system, specifically, the command for executing the generated program. However, further tests need to be carried out to determine other changes that are possibly required.

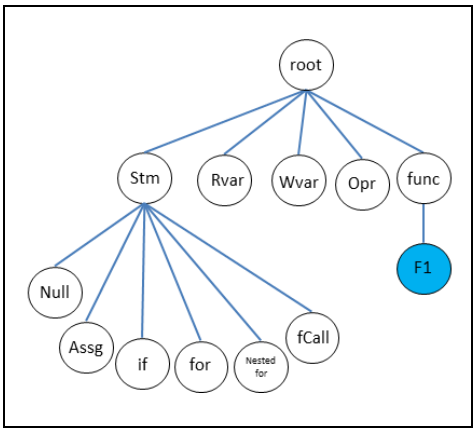

Fig. 5: XML grammar definition structure with an evolved function.

The root is made up of 6 main components. The primitive-statement, which is broken down to core statements such as assignment statement, if-statement and for-loop statement, and support-statement, such as variable and function, are placed on this top level. Each component is defined by a unique rule name with an attribute called 'type' to indicate the conditional status of this rule, i.e., a sequence or a selection rule. The subcomponents are made up of symbols; terminals and non-terminals. The 'F1' in figure 5 above refers to the independently evolved function. Successfully evolved program will be inserted back in the same grammar file and can be called by the main program.

Similar to Withall [11], variables are separated into read–only variables and write-only variables. This separation is necessary to constrain the list indexing. The last component is the FunctionList, and it is made up of a list of functions required by this TGP system, as generated by the called external GP system.

5 Mapping process

This section describes the process of decoding a genome into a phenome using a full syntax grammar definition. We need to scan the grammar for the maximum number of nonterminals on the Right-hand side of the expression to determine the maximum length of the blocks of a genome. What we are looking for is expressed formally as follows:

A grammar G is represented as a 4-tuple:

G = (N , T , P , S),

where :

N is the set of Non-Terminals,

T is the set of Terminals,

P is a numbered set of productions - a set of pairs (PosInt × Production) and

S is the start symbol, S ∈ N.

Unlike normal notation where P is just a set of productions, but here we refer to the productions from a particular non-terminal by ordinal number. The list of all productions defining a particular non-teminal (n) can be discovered using $P(n)=\{ (?,p) : (?,p) \in P \wedge p = (n ::= ...) \}$. All the productions from the same non-terminal are to be numbered sequentially from 0, so $|P(n)| = m \wedge (i,?) \in P(n) \Rightarrow 0 <= i < m$. If $(i,p) \in P$ then p has the form $L ::= R$ and $L \in N$ and $R \in (N \cup T)^*$ [using powerset notation $R \in \wp(N \cup T)$].

Consider one production p written as $L ::= R$

R is a string of values of length r, written $r=/R/$

Each element of R can be referenced by indexing, R_j for j=0 to r-1. The set of indexes is then defined for which R_j is a non-terminal

$NTIdx(R)=\{j : R_j \in N\}$ and the number of non-terminals in R is counted $NTCount(R)= /NTIdx(R)/$.

The set of all counts of non-terminals mentioned on the right hand side of a production P is $NTCounts(P)=\{c:c=NTCount(R) \wedge (?, ? ::= R) \in P\}$. Thus the maximum number of non-terminals among all the productions of a grammar can be found using $max(NTCounts(P))$, where $[m=max(S) \Rightarrow m \in S \text{ and } \forall e \in S, e <= m]$. This defines the number of genes per block (b) in the genotype as $b = max(NTCounts(P))$. Because the genes appear as integer codes, thus if a non-terminal with name n is expected and an integer code i is given then production to be used is $p = (k, n ::= ?)$ where $k = i \text{ modulo } /P(n)/$.

The genotype-phenotype translation algorithm is expressed as follows:

A genotype (GT) is a sequence of blocks ($B_0B_1...B_{s-1}$) for some s. Each block (B) is a sequence of genes ($g_0g_1...g_{b-1}$). Each block records the encoding of one production (p). If *p = (i, L::=R)* then integer codes are given for each R_j in turn for which $R_j \in N$. Note that no codes are given, where $R_j \in T$ because there is no choice - the terminal must be included. The integer code for the non-terminal case chooses which of the relevant productions is to be expanded.

If the encoding process yields less than *b = max(NTCounts(P))* integers, then extra arbitrary genes are added by padding on the right in order to keep all blocks the same length. Genes added in this way are never used in the decoding process but they serve a crucial purpose in the generation/mutation process as explained in Section 3.

6 Experiments and Results

Table 2: Parameter setting for a sort and a swap program evolution.

Parameter	Specification	
	Sort	Swap
Population size	7	7
Selection	Roulette	Roulette
Runs	10	10
Maximum Generations in each run	50,000	50,000
Fitness score target	41	9
Uniform crossover probability	50%	50%
Mutation probability	10%	10%
Machine	Intel 3.00GHz PC with 4GB of RAM, running Windows7	
Input lists	[4 ,3 ,2 ,1], [1,2,55,3], [1,999,2,3], [71,1,2,3], [1,2,33], [100,88,211],[100,1,2], [13,7], [5,55], [10]	[5, 1] , [3, 66]

The initial population is created randomly using various seeds (here we chose the first 10 prime numbers), using parameters setting as in the Table 2 above. Both our 'sorting program' and 'swap program' evolution share the same parameter settings except for the fitness calculation and the test data. Notice that between sort and swap, there is a difference in the maximum fitness score. This score is determined by the number of input lists. There are three experiments carried out as described below.

Experiment 1: The evolution uses a basic grammar (without a swap function). As can be seen in table 4, this technique took longer time to find a fit solution. The best seed value is 2 at 4,407th generation and the worst case is using seed 5, getting a good solution at 36,028th generation.

Experiment 2 : In this experiment, a user-defined swap function is introduced in the grammar. The swap codes are added manually and presented here in BNF:

```
<function>   ::=  <params> "=" <fcall>
<fcall>      ::=  <funcID> "(" "N1" "," "N2" ")"
<params>     ::=  "(" <wvar> "," <wvar> ")"
<funcID>     ::=  "swapnum"
<funcdef>    ::=  "sub" <funcID> "{" <fCode> "}"
<fCode>      ::=  "@_[0,1]" "=" "@_[1,0]"
```

Experiment 3: A pre-processing task is required where the TGP system is executed to generate a swap program. Upon getting an acceptable program, the codes are automatically inserted into the same grammar file. One limitation with this GP system is that the swap program evolution must take place first prior to the sort program evolution. See Section 3.2 for the function structure.

Results : Table 3 presents the experiment result on evolving a swap program. Table 4 shows the comparison of performance between evolution of sorting program without a 'swap' function, with a 'swap' function and with an evolved 'swap' function. The time indicated in the sort with evolved 'swap' function includes the time taken to evolve the 'swap' function, which is independent from the 'sort' function. However, they are by far still beat the 'sort' only algorithm.

Table 3: Swap program evolution.

Seed	Gen	Min
1	1430	00:05
2	10528	00:34
3	11227	00:37
5	5231	00:17
7	2956	00:10
11	47806	02:38
13	2680	00:09
17	20482	01:07
19	918	00:03
23	2331	00:08

The results show a huge improvement in both the number of generations and the time taken when a swap function is used. Only a small increase in the time used by the sort with evolved swap GP program in comparison to the user defined swap

within the grammar for the sorting program GP. The time was devoted to the overhead it takes to evolve the function, adds it to the grammar, executes it and passes its output back to the calling program statement.

Table 4: Comparison of sorting with & without swap.

Seed	Sort without swap		Sort with user-defined swap		Sort with evolved swap	
	Gen	Time (min)	Gen	Time	Gen	Time
1	9114	06:33	52	00:03	52	00:08
2	4407	03:12	875	00:52	875	01:26
3	27830	20:37	473	00:28	473	01:05
5	36028	26:01	668	00:38	668	00:55
7	24400	17:48	135	00:08	135	00:18
11	31384	22:57	1334	01:17	1334	01:55
13	31190	22:56	313	00:18	313	00:27
17	11928	09:00	222	00:13	222	00:20
19	35391	26:25	542	00:33	542	00:36
23	28154	20:44	749	00:47	749	00:55

7 Conclusions and Further work

This paper presents a novel approach to optimise a GP system by giving it an evolved function that it can call repetitively to solve a complex problem. The idea is to provide the GP system with useful function and make it continuously learn the new functions, once provided. The results presented in this paper show a large magnitude of improvement in the fitness evaluation if a useful evolved function is called.

However, the current technique does not make informed choices to which functions may be best to optimise the solution, if there are more functions in the grammar. Therefore, future research will consider determining a mechanism to choose suitable functions for a particular program. We will then extend this technique to do multi-level evolution, for example, swap-sort-reverse evolution and a study of the effect of this technique will then be conducted.

Following this study, the research will focus on application of this technique to evolve the regular expressions. The generated regular expression is used to identify and capture some specific pieces of information, such as, title, location and cost of the course from a training course domain, presented on different variants of HTML web pages.

References:

[1] Angeline, P.J., Pollack, J.B., The evolutionary induction of subroutines. *In Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, 1992, pp. 236–241.

[2] Banzhaf, W., Genotype-Phenotype-Mapping and Neutral Variation: A case study in Genetic Programming. *In Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature*, 1994, pp. 322-332.

[3] Cramer, N.L., A representation for the adaptive generation of simple sequential programs, *In Proceedings of the First International Conference on Genetic Algorithms*, 1985, pp. 183-187.

[4] Harper, R., Blair, A., Dynamically defined functions in grammatical evolution. *In IEEE congress of Evolutionary Computation*, 2006, pp. 9188–9195.

[5] Cooke, J., *Constructing Correct Software*, Springer, 2004.

[6] Koza, J.R., *Genetic Programming*. Cambridge: MA: MIT Press, 1992.

[7] Koza, J.R., *Genetic programming II: automatic discovery of reusable programs*. Cambridge: MIT Press, MA, 1994.

[8] Miller, J.F., Harding, S.L., Cartesian Genetic Programming, *In Proceedings of the GECCO conference companion on Genetic and evolutionary computation*, 2008, pp. 2701-2726.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. *In Proceedings of EuroGP'*, vol. 1802 of LNCS, 2000, pp 121–132.

[10] Siau, N.Z., Hinde, C.J., Stone, R.G., An evolution of a complete program using XML-based Grammar Definition, *In Proceedings of the 4th International Conference on Evolutionary Computation Theory and Applications*, Barcelona, Spain, 2012.

[11] Withall, M.S., Hinde, C.J., Stone, R.G., An improved representation for evolving programs. *Genetic Programming and Evolvable Machines*, 10(1), 2009, pp. 37-70.