Evolving Perl

Mark S. Withall
Department of Computer Science
Loughborough University
Leics. LE11 3TU, UK
m.s.withall2@lboro.ac.uk

Chris J. Hinde
Department of Computer Science
Loughborough University
Leics. LE11 3TU, UK
c.j.hinde@lboro.ac.uk

Roger G. Stone
Department of Computer Science
Loughborough University
Leics. LE11 3TU, UK
r.g.stone@lboro.ac.uk

Abstract

A list of requirements for a genetic programming representation is put forward and a representation separating the genotype and phenotype with a linear genome is presented. The target language for the genetic program is Perl. The mapping process, between the genotype and phenotype, converts blocks of four genes into program statements. This process is context-free and therefore provides inheritable characteristics. The representation is tested by evolving a selection of list evaluation and manipulation functions which are all evolved from the same language subset, with good results.

1 INTRODUCTION

Based on Holland's Genetic Algorithms[3][4] (GA), the idea of evolving programs was first put forward by Cramer in 1985[2]. However, Genetic programming (GP) didn't really become widespread until the work by Koza on the subject[5][6][7].

Most genetic programming representations can be classified into one of three categories; tree, linear and graph[1]. The representations can also be split into those that separate the genotype and phenotype and those that work directly with the actual program.

This paper presents a linear representation for genetic programming, which separates the genotype and phenotype. This allows a simple string (or list) of integers to be used for the genetic manipulation needed to create new generations of solutions. This string is then mapped onto the programming language used (in this case a subset of the Perl[8] language), for the purpose of evaluating the fitness of the solution to the given problem. The mapping process takes blocks of 4 genes from the genotype and converts them to program statements in a context-free manner and, therefore, these individual blocks are inheritable from parent to child. The separation of the genotype and phenotype leaves the system looking more like a traditional genetic algorithm, with the interpretation of the solution string contained within the fitness function.

The first section of this paper sets out a list of requirements for a genetic programming representation. This list of requirements is then used as the basis of a representation which is presented with an example in the following section. Finally, a set of list evaluation and manipulation functions are evolved to test the performance of the representation as part of a genetic programming system.

2 REQUIREMENTS

The following is a list of requirements for a genetic programming representation. It starts with some general requirements for GA representations in general and then moves on to some more specific GP requirements, including requirements of the mapping between the genotype and phenotype in the case where they are separate.

Quick Translation - For every newly created individual in a population, that individual needs to be translated into an executable form for fitness evaluation, in the case where the genotype and phenotype are separate. Therefore the mapping of the genotype to the phenotype needs to be efficient. For example, if there were 500 individuals in a generation and the genetic program was run for 50 generations, the translation of genotype to phenotype would occur 25,000 times. This is a significant proportion of the running time of the genetic program.

Simple Genetic Manipulation - To create a new individual from one or more individuals from the previous generation, it is necessary to use some form of genetic manipulation. This usually takes the form of crossing over the genes of two or more parent individuals and/or performing some kind of random mutation of the new individual. As this process occurs for all or most newly created individuals the representation needs to allow it to be simple and efficient.

Inheritable Characteristics - The major reason why genetic algorithms work is the principle of inheritance. This allows successful characteristics in solutions to be propagated though multiple generations. Therefore it is important that the individuals being evolved are represented in such a way that when a set of genes is passed on to the offspring of the individual, characteristics of the parents are preserved.

Minimal Solution Space - In general, the smaller the solution space, the faster the genetic program will be able to find a solution to the given problem. Alternatively, the larger the percentage of all possible genomes that is made up by good solutions the faster the genetic program will find one. However, the solution space should not preclude possible good solutions to the problem. The size of the solution space, in genetic programming, may be controlled by restricting the syntax and the terminal set available to the genetic program, i.e. the variables, operators and programming constructs to be used.

Maintain Syntactic Correctness - The solution space is restricted also by only allowing syntactically correct programs (phenotypes) to be generated. This rules out a large number of programs which are badly formed.

Limit Execution Errors - As well as errors in the syntax of the programs, other errors such as type mismatches and illegal array indexing can cause problems during the fitness evaluation. These problems need to be avoided where possible. In addition, problems such as infinite loops can disrupt the fitness evaluation process and is especially problematic when the programs are being tested in their natural environment rather than with limited runtime or through emulation.

Allow Scalability - A representation for genetic programming should allow arbitrary length programs to be represented.

Consistent Genotype to Phenotype Mapping - In cases where the genotype and phenotype are separate, it is essential that a given genotype always maps to the same phenotype, in order to result in a deterministic and robust fitness evaluation.

3 REPRESENTATION

This section is divided into four subsections. The first describes the representation for the genotype, the second the corresponding phenotype. The third and fourth subsections describe the mapping between the genotype and phenotype using an example.

3.1 GENOTYPE

The genotype for the genetic program is stored as a simple string (list or array) of integers. The integers are 8 bits (ranging between 0 and 255). Representing the individuals as a string of integers simplifies the process of genetic manipulation, crossover and mutation. The representation is easily scalable allowing arbitrary length individuals.

The following is an example of the representation in its genotype form.

132	24	97	2	...	214

3.2 PHENOTYPE

The phenotype, to which the genotype maps, is a program written in a subset of the Perl programming language. There are various reasons for using the Perl language. Perl is an interpreted language, meaning that it is not necessary to compile the programs that are evolved, for testing purposes. Perl is also self-evaluating, meaning it can create and execute code at runtime. In addition, Perl is untyped (a scalar variable in Perl can hold any scalar data type; string, integer, float, etc.) and declaration of variables is optional. One final feature of Perl which is an advantage in GP is that it has very good error handling and can recover from many error states without crashing.

The phenotype enforces syntactic correctness on the programs. This means that any genotype corresponds to a syntactically correct program. In addition, to minimise execution errors, certain semantic constraints are imposed. For example, to prevent infinite looping, a variable that is the target value of a `for` loop can not have another value assigned to it within the loop. Type mismatches are dealt with automatically by the Perl interpreter.

Figure 1 shows the Backus Naur Form (BNF) representation of the language subset being used. The subset of the language used is fairly small so as to reduce the solution space. Reductions of solution space can also be accomplished by limiting the size of the genome. This BNF representation is easily changeable to support a different target language or to extend the use of the language.

```
(0) <s_list> ::= /* empty */
                 | <s> <s_list>

(1) <s> ::= <v> = <v> ;
          | <v> = <v> + <v> ;
          | <v> = <v> - <v> ;
          | <v> = <v> * <v> ;
          | <v> = <v> / <v> if ( <v> != 0 ) ;
          | if ( <v> <cmp> <v> ) { <s_list> }
          | if ( <v> <cmp> <v> ) { <s_list> }
            else { <s_list> }
          | for <c> ( 0 .. <val> ) { <s_list> }
          | for <c> ( 0 .. <v> ) { <s_list> }

(2) <cmp> ::= ==
            | !=
            | <
            | >
            | <=
            | >=

(3) <v> ::= /* variable */

(4) <val> ::= /* value */

(5) <c> ::= /* counter variable */
```

Figure 1: Subset of Perl language being used

3.3 MAPPING THE GENOTYPE TO THE PHENOTYPE

The method of converting the genotype into the phenotype is fairly simple, allowing for quick translation.

As can be seen from the BNF (Figure 1) description of the language, each statement can be defined by at most four genes (the sublists of statements are coded as separate blocks), one to define the statement type and at most three for variables and operators. The genome is therefore split into groups (or blocks) of four genes where each group of four represents a statement. This method allows some redundancy in the representation and also means that mutations will only affect the statement that the mutated gene is in. Most importantly, this means that each block of four genes is context-free and will always evaluate to the same statement, so if it is inherited by an individual from a parent, the same statement will appear in both the parent and child.

The first gene in the block determines the type of statement, ten in this case. The subsequent three genes define the variables, values or comparison operators that make up the statement. Table 1 shows the available statements and their corresponding gene value. The last statement is really just a marker for the end of a statement list. If it appears at the top-level of the program it is ignored, however, within an `if` or a `for` statement, it defines the end of the sublist of statements. As the gene values can range from 0 to 255, it is necessary to take modulo 10 of the gene value to determine the corresponding statement.

Table 1: List of available statements

VALUE	STATEMENT
0	Assign
1	Add
2	Sub
3	Mul
4	Div
5	If
6	IfElse
7	ForVal
8	ForVar
9	End

Most of the statements require one or more variables. The variables are usually predefined in the wrapper code (see the example individual, section 3.4) and added to a list of variables. This means that the number of variables is variable. When a gene represents a variable, the modulo number of variables is used on the gene value to determine the corresponding variable. Table 2 shows an example list of variables which may be in use for an individual, this basic list is problem specific and is defined in the wrapper for the problem (see 3.4). Additional variables can also be added during the mapping, for objects such as loop counters.

In the `if` statement a comparison operator is required. There are six different comparison operators used, which are shown in Table 3. The gene representing the comparison operator is converted using the same modulo method as above.

In the `for` statement an integer value or variable is required as the upper bound for the loop. This value is

Table 2: Example list of variables

VALUE	VARIABLE
0	`$n`
1	`$zero`
2	`$res`
3	`$c0`

Table 3: List of comparisons

VALUE	COMPARISON
0	`==`
1	`!=`
2	`<`
3	`>`
4	`<=`
5	`>=`

arbitrarily constrained to be an integer between 0 and 19 in the later example (3.4) and therefore the modulo 20 of the gene value is written into the `for` statement (see Table 5), this is just to stop programs with excessive run-times being generated. In practice, this modulo value can be based on the input data for the fitness testing. In addition, the loop counter variable is added to the list of variables available and an indexed list element (such as `$list[$c]`) if appropriate.

This process of conversion means that a genotype will always be mapped to the same phenotype and the generated program will always be syntactically correct.

3.4 EXAMPLE INDIVIDUAL

As an example of the conversion from the genotype to the phenotype using the above representation, a possible solution to the problem of *factorial* is given.

Table 4 shows the genome as a list of integers. The values that are not used for the conversion have been replaced with an X.

Table 4: Example genome

38	21	X	X	25	55	25	73
13	38	66	87	49	X	X	X

Each program statement is generated from four genes. Table 5 show the four groups of four genes that make up the program. The first gene from each group represents the statement type. The modulo 10 of this value gives a number between 0 and 9 and this corresponds to a statement type from table 1.

In the case of the first group, the first gene value is 38, 38 modulo 10 gives the value 8 which corresponds to a `ForVar` statement. The second value, 21, corresponds to the target variable in the `for` loop. The last two genes are redundant for this statement, therefore the statement looks like `for $c0 (0..($n%20)){`. The variable `$c0` is then added to the list of variables, that is an arbitrary choice and subsequent loop variables will be named with increasing numerical suffixes. The remaining three groups of genes are handled in the same way as above. Table 5 shows all of the statement conversions for the genome.

Finally the wrapper code, shown in Figure 2, is added to provide the variables for the problem and the output of the solution. Constant values are handled as variables so that it is not necessary to choose either a variable or a value. The final program, after the completed conversion, is shown in Figure 3. Note, the program will only work correctly for input values up to 19.

```
# Header
$n = ARGV[0];
$zero = 0;
$res = 1;

# Main Code

# Footer
print "$res";
```

Figure 2: Wrapper for example

```
# Header
$n = ARGV[0];
$zero = 0;
$res = 1;

# Main Code
for $c0 (0..($n%20)) {
  if($c0 != $zero) {
    $res = $res * $c0;
  }
}

# Footer
print "$res";
```

Figure 3: Example program to perform 5 factorial

Table 5: Conversion of genotype to phenotype

GENES	MOD	STMT	CODE
38,21,X,X	8,0,X,X	For	`for $c0 (0..($n%20)){`
25,55,25,73	5,3,1,1	If	`  if($c0 != $zero){`
13,38,66,87	3,2,2,3	Mul	`  $res = $res * $c0;`
49,X,X,X	9,X,X,X	End	`} }`

4 EXPERIMENT

The following experiment tests the above representation with the target of evolving a selection of list evaluation and manipulation functions. This section first describes the functions being evolved, the fitness functions used to test individuals during evolution and the experimental procedure used for testing. Finally the results of the experiment are given and a selection of evolved programs are shown.

4.1 FUNCTIONS BEING EVOLVED

The following list evaluation functions are evolved:

Max List - Find the largest element from a given list of integers.

Min List - Find the smallest element from a given list of integers.

Sum List - Find the sum of all elements from a given list of integers.

Average List - Find the average value of all elements from a given list of integers.

In addition, the following list manipulation functions are evolved:

Reverse List - Reverse the order of the elements from a given list of integers.

Sort List - Sort a given list of integers into numerical order from smallest to largest.

4.2 FITNESS FUNCTIONS

The following sections describe the fitness functions used to evolve the above set of functions. Arbitrarily, seven arbitrary lists of varying sizes were used as test inputs, except for the sort function where sixteen were used.

4.2.1 Max List

Each individual is given each of the test lists as an input and the output value returned by the individual is compared to the expected output. If the output is correct the individual receives 10 points of fitness, otherwise no points are given. An individual can therefore have a maximum fitness of 70. The actual maximum fitness for an individual is 71 as a starting fitness of 1 is given to each individual so that even the worst individuals have a possibility of reproducing.

4.2.2 Min List

As with max list, for each correctly evaluated list the individual receives 10 fitness points. Again, each individual can have a maximum fitness of 71.

4.2.3 Sum List

For sum list, the absolute difference between the expected result and the result returned by the individual was taken as the fitness value. The sum of these differences, for all seven tests, was taken as the overall fitness. This value was then normalised by subtracting it from 5000 (this value is based on the values in the test lists), making higher fitness better (and 5000 optimal). Any value greater than 5000 was normalised to 1.

4.2.4 Average List

The fitness test for average list was carried out in the same way as that of sum list, with the normalised fitness value being between 1 and 5000.

4.2.5 Reverse List

For reverse list, each of the individuals was given 100 fitness points for each of the returned lists which were of the correct length and a further 10 points for each of the list elements which were in the correct position. For the chosen lists a total fitness of 961 points were available, with a minimum score of 1.

4.2.6 Sort List

As with the reverse list fitness, 100 points were given for each returned list of the correct length and a further 10 points for each element which was in the correct position. In addition, 10 points were given for each pair of elements in the list which were in the correct order. This gave a maximum possible fitness of 1691 and, again, a minimum fitness of 1.

4.3 EXPERIMENTAL PROCEDURE

The following sections describe the hardware and software used for the experiments, the parameters used for the genetic programs and the wrapper code used for the individuals being evolved.

4.3.1 Hardware And Software

The genetic program was written using Perl v5.6.1 and run on a PIII 866 PC with 128MB of memory, running Windows 2000.

4.3.2 GP Parameters

All of the above functions were evolved with population sizes of 7 and mutation rates of 1 gene in 20. All of the functions were evolved with a fixed-length genome size of 40 genes apart from the sort function which had a genome size of 60 genes as preliminary tests suggested 40 was too restrictive. The genetic operators used for the experiments were a uniform (every-point) crossover and single-gene random mutation and all experiments were initialised with a seeded randomly generated population.

All of the tests were run ten times with a fixed set of random number generator seeds (1, 2, 3, 5, 7, 11, 13, 17, 19, 23), this was so the experiments were repeatable. As the language subset was the same for all of the functions, the only variable in the experiment was the fitness test.

4.3.3 Wrapper Code

Figure 4 shows the general structure for the wrappers for the list evaluation functions. The additional 'temp' variable is only used for the average list function. The +1 to get the size of the list is beacuse `$#list` returns the index of the last element in the array and not the array size. In addition to the variables in the wrapper, the indexed variable `$list[0]` is also added to the list of variables.

Figure 5 shows the general structure for the wrappers for the list manipulation functions. Both reverse list and sort list have been given three 'temp' variables to work with. All the list manipulation has to be done inplace as there is only one list variable available during the evolution.

```
# Header
my @list = @_;
my $size = $#list+1;
my $res = 0;
my $tmp = 0;  #optional

# Evolved Code

# Footer
return $res;
```

Figure 4: General structure of the wrapper for list evaluation

```
# Header
my @list = @_;
my $size = $#list;
my $tmp1 = 0;
my $tmp2 = 0;
my $tmp3 = 0;

# Evolved Code

# Footer
return @list;
```

Figure 5: General structure of the wrapper for list manipulation

4.4 RESULTS

Table 6 gives the complete set of test results. The time and generation given are to reach the optimal solution. In all cases the optimal solution was reached as defined by the fitness function. Figures 6 to 8 show example solutions to some of the function evolutions. Redundant code has been removed from the solution programs, for added clarity, in cases where it was obvious that the code had no effect, e.g. statements such as `$x = $x;` or `if($x != $x){...}`. No code was removed that couldn't be removed easily in an automated way. The exception is Figure 7, the sum list solution, which has been left in its original state as an example.

5 SUMMARY AND CONCLUSIONS

The following list describes how the representation presented fits in with the requirements set out.

Table 6: Results for list function evolution, S=seed, T=time and G=generations

S	MAX		MIN		SUM		AVERAGE		REVERSE		SORT	
	T	G	T	G	T	G	T	G	T	G	T	G
1	1m48s	3427	33s	1090	10s	275	38s	408	1h17m41s	128090	8h09m11s	30779
2	2m12s	2777	2m24s	2368	14s	239	4s	68	35m36s	36054	56m43s	5467
3	6m41s	15454	24s	497	7s	202	4m20s	2500	1h12m27s	72721	44m04s	5760
5	25s	756	5m18s	7987	3s	68	2m13s	2972	3h00m06s	260092	27m19s	6036
7	42s	917	1m08s	1431	5s	146	31s	604	19m15s	27845	58m19s	10997
11	18s	593	17s	480	3s	93	3m32s	4832	53m18s	92143	11m10s	5278
13	1m48s	3322	2m53s	5666	3s	143	45s	1057	2h56m34s	205277	1h47m47s	25860
17	14s	462	7s	202	10s	143	1m58s	1156	1h08m52s	88653	2h53m55s	12486
19	2m05s	4467	1m09s	1381	1s	28	16m46s	5205	2h46m31s	200843	5h13m09s	17320
23	40s	1292	2m43s	4657	3s	87	12s	196	2m03s	3776	2h07m14s	42804

```
# Header
my @list = @_;
my $size = $#list+1;
my $max = 0;

# Evolved Code
$max = $list[0];
for my $c0 (0..($size-1)%10) {
  if($list[$c0%10] > $max) {
    $list[0] = $list[$c0%10];
    $max = $list[0];
  }
}

# Footer
return $max;
```

Figure 6: Max List, Seed 2

```
# Header
my @list = @_;
my $size = $#list+1;
my $sum = 0;

# Evolved Code
for my $c0 (0..($size-1)%10) {
  $sum = $list[$c0%10] + $sum;
  if($list[$c0%10] != $sum) {
    $c0 = $list[$c0%10] + $c0;
    for my $c1 (0..5) {
      if($list[$c0%10] >= $c1) {
        for my $c2 (0..5) {
        }
      }
    }
  }
}

# Footer
return $sum
```

Figure 7: Sum List, Seed 5

Quick Translation - The representation implements a fairly direct mapping between the genotype and phenotype, therefore the translation process can be performed in linear time.

Simple Genetic Manipulation - As the genome is linear and there are no special requirements for crossover or mutation, the genetic manipulation operations are simple.

Inheritable Characteristics - The representation implements statements as blocks of 4 genes with each block being context-free. Therefore, in the mapping between the genotype and phenotype, a block of 4 genes always maps to the same statement in both parent and child.

Minimal Solution Space - As the BNF represents only a small subset of the Perl language and only a small number of variables are used for each problem, the solution space is kept fairly small.

Maintain Syntactic Correctness - The mapping from the genotype to the phenotype ensures that all lists of integers map successfully to a Perl program and all programs generated are syntactically correct.

Limit Execution Errors - As Perl is fairly robust in its error handling and all programs generated are syntactically correct, there should be no execution

```
# Header
my @list = @_;
my $size = $#list;
my $tmp1 = 0;
my $tmp2 = 0;
my $tmp3 = 0;

for my $c0 (0..$size%20) {
  for my $c1 (0..$size%20) {
    $tmp3 = $list[$c0%20];
    for my $c2 (0..$c1%20) {
      if($list[$c2%20] <= $list[$c0%20]) {
      }
      else {
        $list[$c0%20] = $list[$c2%20];
        $list[$c2%20] = $tmp3;
      }
    }
  }
}

# Footer
return @list;
```

Figure 8: Sort List, Seed 11

errors.

Allow Scalability - The genotype in the representation is a list of integers which is infinitely extensible and as each block of 4 genes maps to a statement in a context-free manner, the programs can be of an unlimited size.

Consistent Genotype To Phenotype Mapping - The list of integers always maps to the same program.

In general, the results of the list function evolution experiments were good. Some of the sort list functions took a long time compared to the number of generations, this is mainly due to a large number of nested `for` statements causing long run times during the fitness testing. This problem can be easily overcome in future experiments by limiting the number of nested `for` loops.

An additional cause of slow run times is the genetic programming system being implemented in Perl. If the system were implemented in C the run times would also decrease.

Long run times for the reverse list functions could possibly be accounted for by poor design of the fitness test set and scoring method.

In general, the solutions generated were similar to those which might be generated by a human programmer confronted with the same syntax constraints.

In summary, a list of requirements for a genetic programming representation was put forward and then a representation which followed the requirements was described. This representation was then tested on a set of list evaluation and manipulation problems and produced good results.

Acknowledgements

Thanks to everybody and all their friends. In addition, thanks to Nortel for their support, both financial and intellectual, during the project.

References

[1] Banzhaf W. Nordin P. Keller R.E. & Francone F.D. (1998). Genetic Programming - An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann.

[2] Cramer N.L. (1985). A Representation for the Adaptive Generation of Simple Sequential Programs. In Grefenstette J.J., editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187.

[3] Holland J.H. (1973). Genetic Algorithms and the optimal allocation of trials. In *SIAM Journal on Computing*, 2(2):88–105, June.

[4] Holland J.H. (1992). Adaption in Natural and Artificial Systems. MIT Press, second edition.

[5] Koza J.R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press.

[6] Koza J.R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press.

[7] Koza J.R. Andre D. Bennett F.H. & Keane M. (1999). Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufmann.

[8] Wall L. Christiansen & Schwartz R.L. (1996). Programming Perl. O'Reilly & Associates, Inc., Second Edition.