An Improved Representation For Evolving Programs

M.S. Withall (m.s.withall@lboro.ac.uk), C.J. Hinde (c.j.hinde@lboro.ac.uk) and R.G. Stone (r.g.stone@lboro.ac.uk)
Department of Computer Science, Loughborough University, Loughborough, Leics. LE11 3TU, England

Abstract. A representation has been developed that addresses some of the issues with other Genetic Program representations while maintaining their advantages. This combines the easy reproduction of the linear representation with the inheritable characteristics of the tree representation by using fixed-length blocks of genes representing single program statements. This means that each block of genes will always map to the same statement in the parent and child unless it is mutated, irrespective of changes to the surrounding blocks. This method is compared to the variable length gene blocks used by other representations with a clear improvement in the similarity between parent and child. In addition, a set of list evaluation and manipulation functions was evolved as an application of the new Genetic Program components. These functions have the common feature that they all need to be 100% correct to be useful. Traditional Genetic Programming problems have mainly been optimization or approximation problems. The list results are good but do highlight the problem of scalability in that more complex functions lead to a dramatic increase in the required evolution time.

Keywords: Genetic Algorithms, Genetic Programming, Representation

1. Introduction

The paper is organised into two major divisions, an introduction and history with a discussion of the requirements of a representation for Genetic Programming and the contribution to Genetic Programming that the representation gives.

1.1. Overall Goals of the paper

In order to set the paper into context there are some overall requirements that are necessary for a Genetic Programming system. The descriptions of the various representations that are in use may be seen in this context, and should make the aim of the paper clearer. The requirements presented in Section 2 refine these aims and make them more specific. The overall requirements are as follows:

Operator independence — The representation of the phenome should be independent of the genetic operators. This would have the

benefit of allowing other problem solving paradigms, simulated annealing, Monte Carlo, etc. to be applied to the genome regardless of the structure of the phenome.

Minimal search space — The representation should be interpreted in such a way that it maps closely onto the search space. This reduces the search space represented by the genome. If the representation is optimally coded then every genome would represent a valid phenome. If the representation is such that not all genomes convert to a valid phenome then the nearest valid phenome must be selected and this requires a repair stage.

Inheritable characteristics — The representation should maintain visible inheritance of phenome characteristics from parent to child. If this was a characteristic that made the parent successful then it would be passed on to a child. An optimally coded tree, say using Huffman codes, would be radically changed if one bit near the start of the tree were changed.

1.2. Brief History

One of the earliest known pieces of research on using evolution to create computer programs was that of Friedberg *et al* [10, 11]. Although the word 'evolution' does not appear in either paper the intent to simulate evolution was plainly in the minds of the researchers [7]. Friedberg *et al* adopted the task of generating a set of machine language instructions that could perform relatively simple calculations (in this case adding the numbers in two data locations). The work of Fogel, Owens and Walsh [8], which evolved finite state machines, and the work of Holland [18] and others on learning classifier systems could also be classed as Genetic Programming.

Possibly the first work that explicitly used Genetic Algorithms to generate programs was that of Cramer [6] in 1985. This was closely followed by the work of Fujiki *et al* [12, 13], who used the method to solve the prisoner's dilemma, and the work of Hinklin [17]. All of the above used a tree representation for their programs.

The work that popularised the area (which become known as 'Genetic Programming') was that of John Koza, initially with his 1989 paper [24] and more so with his three epic works [25, 26, 27]. This work also used the tree representation with the target programming language LISP (the same as Fujiki *et al*).

Since the work of Koza, a vast amount of research has been done on the area of Genetic Programming. Banzhaf *et al* describe three types

of representation: tree, linear and graph [3]; although other approaches such as Kantschik's *linear-tree* [20] and *linear-graph* structures [21], also exist. Some of the main systems that have been developed are looked at in more detail in the following sections.

1.3. Tree Representation

Most of the early work on Genetic Programming was done using a tree representation and the method is still widely used today.

The tree representation is close to the phenome and as such is efficient to convert to the program form. However, the crossover and mutation operators are more specialised than the crossover and mutation operators used in the linear representation, (Section 1.4). If the choice of crossover point is random between the nodes then the crossover points will be at the lower level of the tree as there are many more nodes at the leaf nodes unless the choice points are biased. In an n-ary tree it would be possible to explore the tree from the root and make an n+1 way decision whether to explore the one of the n branches, or choose as a crossover point. The weights chosen control the behaviour. Crossover in the linear representation described in Section 1.4 is independent of the phenome representation in contrast to crossover in the tree representation.

One of the first attempts that explicitly used Genetic Algorithms to evolve programs was by Cramer [6]. Cramer demonstrated an adaptive system for generating short sequential computer functions (in his paper two-input, single-output multiplication functions were evolved). The initial representation for the programs was a list of integers, which were then decoded to produce a well-formed program. The advantage of this approach was that any list (of sufficient length and with the relevant constraint on the size of the integers) could be used to generate a well-formed program. The problems were that infinite loops can be generated by the auxiliary statements and, more seriously, the semantic-positioning of an integer-list element is extremely sensitive to change. This illustrates the repair problem introduced in Section 1.1.

To address these problem, a modified version was created, which had a tree-like structure. However, to avoid the problems of 'catastrophic minor changes' the mutation and crossover operators had to be constrained. In this case, the mutation operator can only change leaf statements or non-leaf statements that only have arguments that are leaf operators. For the crossover, subtrees are swapped between two parents.

Cramer also cited the work of Smith [39], which pointed out that a major problem is that of 'hand-crafting' the fitness evaluation function

to give partial credit to functions that exhibit behaviour similar to that desired, without actually performing the desired task.

Koza first introduced his method of Genetic Programming in 1989 [24]. This work was then developed [25, 26, 27].

For Koza's Genetic Programming, the programs are represented as parse trees. The language LISP was used, as a subroutine in LISP (or s-expression) is essentially a parse tree expressed in a linear fashion. For Genetic Programming, the user defines all the functions, variables and constants which can be nodes in the parse tree.

Due to the complex nature of the structure of the genomes (LISP s-expressions), the genomes cannot be easily generated randomly for the initial population. The individuals in the initial population must be carefully constructed to preserve syntactic correctness. In addition, the genetic operators used cannot be the simple versions of crossover and mutation used in the linear form described in Section 1.4. Instead, mutation is accomplished by picking a random node in the tree and replacing the subtree with a randomly generated (but syntactically valid) subtree. The crossover operator exchanges subtrees from two parent individuals; see above for comments about the random nature of crossover point selection.

Montana [28] was one of the first people to look at the problem of function and variable typing in Genetic Programming (although it had been mentioned by Koza [25]). The approach built directly on top of the approach which Koza used. Another approach was presented by Perkis [35] using multiple stacks, one for each type. The use of strong typing in GP becomes essential when the target language is *e.g.* C or pascal, as type mismatches would cause the programs to fail to compile; see Sections 1.1 and 2.

The method presented by Montana also introduced handling of run-time errors, whereas the method used by Koza forced all functions to return valid values *e.g.* the protected divide function returned 1 when dividing by zero, rather than an error.

One of the first people to use a context-free grammar as the basis of their representation was Whigham [43], who used the tree structure. One of the main advantages of using the context-free grammar is that it allows the method to be applied to any contemporary programming language [15]. The grammar also allows variable typing to be incorporated easily.

Whigham also used the idea of 'bias' (structuring the grammar in such a way as to improve the chances of creating good programs). This is equivalent to including extra knowledge about the problem *e.g.* if it is known that the program should start with an `if` statement. Rather than manually adjusting the grammar, Whigham modified the

grammar during the evolution based on analysis of fit individuals. Each generation some new individuals were created from the updated grammar and incorporated into the population. In addition, Whigham used weighted production rules to make selection of good rules more likely. This weighting was calculated as the new production rules were created.

Angeline et al. [1] made a significant contribution to scalability using tree representations in their work on induction of subroutines using a Graph representation. Essentially they removed branches of the evolving trees and stored them for use as single nodes. They take this further as indiscriminate compression could have undesirable side effects and incorporate an expansion operator. They state that the complementary nature of the compression and expansion operators implements a form of iterative refinement. The random selection of a subtree for compression provides no guarantee that the selected subtree will be an above average schema. It is more likely that it will be either a portion of a useful schema or simply of no import at all. By periodically replacing a copy of the compressed subtree back into the population, they provide the chance to capture a better version of the schema at a later time.

1.4. Linear Representation

The linear representation is closer to the Genetic Algorithm as described in Holland [18]. Separate crossover points in the parents allow variable length chromosomes. There are two consequences of extending the chromosome: the first is that programs of arbitrary length may be produced and the second is that it precludes uniform crossover [40] and other variants. Uniform crossover is reported to be very efficient. Wineberg and Oppacher [44] use a fixed length genome but pad the genome with introns. Introns are genes that have no effect and in effect are just padding. They may arise as a result of mutations and errors but as they have no effect on the phenome they can be removed and as such are padding. Introns are argued to be beneficial to the search process [45], so although introns do not code for any aspect of the phenome they affect the evolutionary process. Recent work on introns indicates that the presence of introns can prematurely stabilise the evolution [19]. Both methods generally suffer from the problem that the context of a gene changes its effect on the overall phenome.

Perhaps the best example of a linear representation that combines the advantages of the linear representation with the advantages of minimal translation is that of Nordin [30], which directly manipulates machine code by casting a binary string onto a function. This binary string is the genome and so this mechanism maps the genome directly onto the phenome and also avoids any compilation steps in the system.

This representation is both a linear genome and also maps directly onto the phenome. The search space that Nordin's system works within is very large and there is almost no opportunity to apply a hierarchical design strategy to it.

In 1993, Banzhaf [2] introduced a method of Genetic Programming based on traditional Genetic Algorithms. The method introduced mechanisms like transcription, editing and repairing and was applied to the problem of the prediction of sequences of integer numbers. This is one of the first methods to go back to using a linear genome, since Cramer [6] rejected the idea in favour of a tree representation.

Banzhaf starts out with a population of binary strings which are subsequently interpreted as programs by using a table specifying which binary code corresponds to which element from the set of functions and terminals available. The generated program can not necessarily be guaranteed to be a working program. After the binary strings have been translated into the programming language, the resulting code segments are checked to see if they are syntactically correct and any errors are repaired.

The work is extended by Keller and Banzhaf [23]. Here they borrow heavily from molecular biology and only use the mutation operator for genetic manipulation. They also show that their method can map to an arbitrary context-free language.

Perkis [35] presented yet another approach to the evolution of programs that does not require specialist genetic operators. In this approach the genome is a sequence of functions and terminals. Each element in the sequence is evaluated in turn. If the element is a function it takes the necessary number of arguments off a stack and puts its output back onto the stack. If there are not enough values on the stack then the function is ignored. If the element is a terminal it is pushed onto the stack. As there are no syntactical constraints on the sequences, they can be treated in the same way as a traditional binary string for a Genetic Algorithm.

One limitation of the method presented is that it has no mechanism for branching. In addition, it would be difficult to generate a program in a specific target language using this method.

Paterson and Livesey [33] continue on from the work of Banzhaf, by introducing a method that converts a linear genome (in this case a string of integers) into a program. Unlike Banzhaf, however, there is no need for a repair stage as the list of integers maps directly onto a BNF definition of the language subset by recursively replacing all non-terminals with the production rule that corresponds to the next integer in the genome. Like Koza and others, Paterson and Livesey initially use LISP as their target language but in later work they use C [34],

showing the advantage of the method being language independent. One disadvantage of mapping the list of integers to the BNF is that there is no guarantee that a complete program will be generated. There may be unresolved non-terminals when the string of integers runs out. Other methods must then be used to fill in the missing data. One interesting experiment conducted by Paterson and Livesey was to compare two grammars that represent the same language subset. This highlights the difficulty of specifying the language subset in the most appropriate way for the problem.

Ryan, Collins, and O'Neill [37] present a method that is similar to Paterson and Livesey, but with a much simpler mapping between the genome and phenome. Ryan, Collins and O'Neill still use the BNF representation, although they use a binary string instead of a list of integers.

The major disadvantage of their method is from the point of view of inheritance of characteristics. When a gene is passed on to a child individual there is a very high chance that the gene will not represent the same value. One change early in the genome can change the entire path through the grammar and hence the child individual will have very little resemblance to its parent.

Despite this drawback, Grammatical Evolution has been applied to a variety of different problems by the original authors [32, 31, 38]. In addition, the method is developing a following around the world [16, 22].

Other work on linear and grammar-based representations for Genetic Programming has been done by Freeman [9] and Ross [36].

1.5. Graph Representation

The graph representation, see Figure 1, is a natural development of the tree representation. Parse trees are hierarchical so using graphs is more general and is able to be applied to a wide variety of problems. Graph representation follows the trail of bringing the genome closer to the Phenome. A classic system is PADO, *Parallel Algorithm Discovery and Orchestration* [41].

Niehaus and Banzhaf [29] describe a directed graph based Genetic Programming representation called GGP. They have an abstraction mechanism which addresses scalability by using subfunctions.

1.6. Summary of Contents

The following sections describe the representation for evolving programs, which addresses the weaknesses of the tree and linear representations in detail while keeping their benefits. Graph representations are included in the comments about trees as the conclusions are similar.

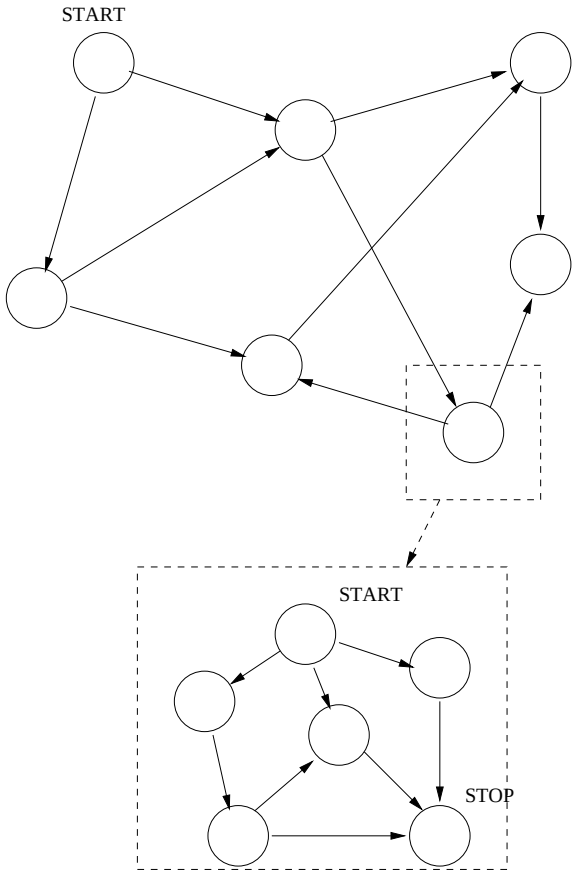


Figure 1. The representation of a program and a subprogram in the PADO system. Each node consists of: Number, Action, Branch, Arc1, Arc2, Branch Constant

2. Requirements for a New Representation

This section takes the requirements for a Genetic Programming representation introduced in Section 1.1. These requirements are largely based on more detailed analysis of those representations.

Quick Translation — In the case where the genome and phenome are separate, every newly created individual in a population needs to be translated into an executable form for fitness evaluation. Therefore, the mapping of the genome to the phenome 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 the genome to phenome would occur 25000 times. This can be a significant proportion of the running time of the

Genetic Program. Even if caching is used to avoid evaluating duplicate genomes [5] the work involved is significant. In the work by Koza [25] and others who use languages such as LISP, this is not a problem as the genes are stored directly as program fragments. In the work of Banzhaf [3] there is the additional complication of the repair of badly structured individuals, which can be costly in terms of run time.

Simple Genetic Manipulation — To create a new individual from one or more parent individuals it is necessary to use some form of genetic manipulation. This usually takes the form of combining the genes of two or more parent individuals and/or performing some kind of random mutation on 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 — One of the main reasons why Genetic Algorithms work is the principal of inheritance. This allows successful characteristics in individuals to be propagated through 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. If the genome and phenome are to be separated, there needs to be a fairly direct relationship between the two in order for phenotypical characteristics to be inheritable. Koza [25] has this property because the genome and phenome are the same and, therefore, the child phenomes are constructed directly from parts of the parent phenomes. For representations such as Grammatical Evolution [37], there is not a direct mapping between the parent and child phenomes and, therefore, crucial characteristics from the parents can be lost when the child is generated (see Section 4).

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 correspond to good solutions the faster the Genetic Program will find one. However, the solution space should not unduly restrict the range of possible solutions to the problem. The size of the solution space may be controlled by restricting the language subset available. This may be part of the use of the representation rather than the representation itself. For example, in Grammatical Evolution [37] the solution space is dependent on the BNF grammar given to the system by the user. An additional

factor that can have an effect is the shape of the solution space. If, for example, there are many local minima (or maxima) then it may be a more difficult space to search for the Genetic Program.

Maintain Syntactic Correctness — The solution space is also restricted by only allowing syntactically correct programs (phenomes) to be generated. This rules out a large number of programs that are badly formed. Representations, such as Koza's [25], use genetic operators which maintain syntactic correctness. In this the genetic operators need to be suitable chosen based on the target language.

Limit Execution Errors — As well as errors in the syntax of the programs, other errors that occur at run-time can cause problems during the fitness evaluation. Montana [28] had problems with his system, in that very few of the initially generated population of programs were correctly typed. It therefore took a long time to find initial viable programs before they could be improved to solve the task set. These problems need to be avoided where possible. In addition, problems such as infinite loops can disrupt the fitness evaluation process and are especially difficult to deal with when the programs are being tested in their natural environment rather than with limited runtime or through emulation.

Consistent Genome to Phenome Mapping — In cases where the genome and phenome are separate, it is essential that a given genome always maps to the same phenome, in order to result in a deterministic and robust fitness evaluation. Paterson and Livesey [33] suggested that one possible approach in their representation, when the genome ran out of genes in the mapping process, was to randomly generate the rest of the phenome. This approach is not very good from the perspective of inheritable characteristics.

To summarise, a representation is required that has a separate genome and phenome, where the genome is a simple representation for genetic operators, which has no special constraints and the phenome is a program in the target language. Every genome should only map to syntactically correct programs in a concise language subset and where possible the language subset should be restricted to avoid problems such as infinite loops. Most importantly, the mapping between the genome and phenome must be simple and fairly direct, so that the characteristics in the child phenome can be inherited from the parent phenome during the genetic manipulation.

3. Description of Representation

The representation satisfies the requirements given above. Primarily the description focuses on the Genome, which is simple, and the phenome which is derived as a result of the translation process.

3.1. Genome

The genome for the Genetic Program is stored as a simple string (or list) of integers. The integers used in all the examples in this paper are 8-bit (ranging between 0 and 255) but any size of integer is acceptable as long as it satisfies the requirements of the mapping process (see Section 3.3). Representing the individuals as a string of integers simplifies the process of genetic manipulation, crossover and mutation.

3.2. Phenome

The phenome, to which the genome maps, is a program written in a subset of some language, in the case of the examples in this paper, Perl [42]. 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 fitness evaluation. Perl is also capable of executing program statements that are generated during the running of a program, which means that the evolved programs don't have to run externally to the GA. One final feature of Perl, that is an advantage in GP, is that it has a good error handling and recovery, so if an evolved program does not work properly it won't affect the rest of the GP.

The subset of the language chosen for a particular problem can easily be designed with certain semantic constraints, such as avoiding infinite loops. For example, only including restricted 'for' loops, where the counter variable can not change within the body of the loop.

3.3. Mapping from Genome to Phenome

The mapping between the genome and the phenome ensures that all genomes map to a syntactically correct program in the required language.

The mapping starts by dividing the genome into fixed-length blocks of genes, each of which represents one program statement. The length of the gene blocks is dependent on the statement type that requires the most information. Each block is interpreted independently of the others. So, two identical gene blocks in different places in the sequence will be interpreted the same way. It can easily be seen that if all of the blocks are the same size, and they are interpreted independently, then

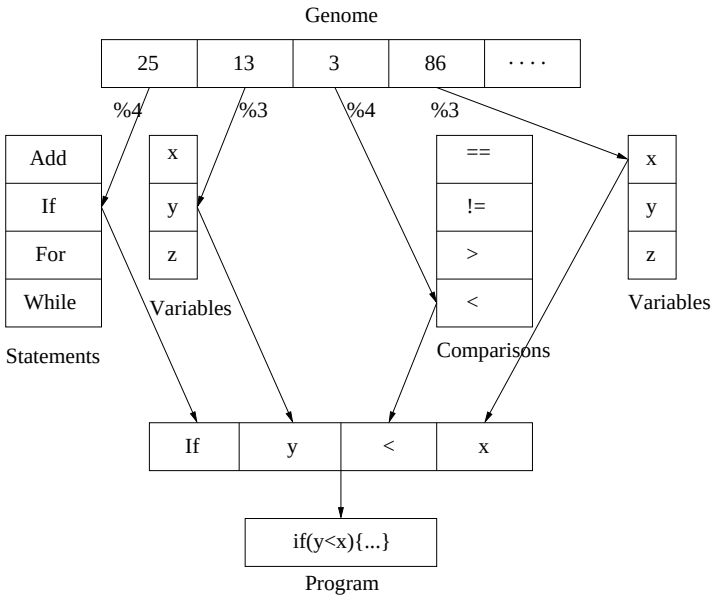


Figure 2. Example mapping from individual gene block to program statement

when a block is inherited by a child individual it will be in the same place and therefore be interpreted the same way (assuming fixed-length genomes). This ensures the inheritance of phenotypical characteristics even though a separated genome and phenome is being used. In addition, this method of translating the genome to the phenome ensures that a complete program will be generated without running out of genes, as would be the case in the work of Paterson and Livesey [33] or Ryan *et al* [37].

The first gene in each block represents the type of statement. The statement type is decided by taking the modulo of the gene value and the number of different program statements. For example, if there are four statement types and the gene value is 23 then $23 \equiv 3 (\mathrm{mod}\ 4)$, so the fourth statement (index number 3) would be chosen.

Each statement type uses the remaining genes in different ways. For example, an 'Addition' statement would require one variable to assign the result to and two variables to add together. Any remaining genes in the block are redundant. Figure 2 shows an example of the mapping from a gene block to a program statement. Table I shows an example genome in blocks of 4 values and Table II shows a list of possible statements, the use of the remaining genes and the form in which the statement is presented in the target language. Note that the indices associated with the genes in the genome are taken modulo the

size of the list to which they refer. For example, the first value 28 is associated with the statement type; there are 5 statement types and so 28 refers to statement type 3, a 'For' statement.

3.4. Extensions

In addition to the basic mapping, there are a few useful additions to be able to evolve reasonable programs. The first is the need for an ability to provide nested structures, such as looping and branching, without losing the inheritance features of the current mapping. Here it is achieved by having a statement type *e.g.* a 'For' statement, which has a corresponding 'End' statement. All statements in between these two statements are then nested within the loop or branch. Although a mutation to the statement type, of the loop or branch statement, would change the structure of the subsequent code, the meaning of the individual statements will still be preserved. Any remaining nested structures that haven't been terminated when the end of the genome is reached can then be automatically terminated.

An additional feature that can be used, is to distinguish between variables that are read-only and read/write, so that any variables that should not be changed cannot be assigned new values. In practice, two sets of variables are stored, one is all of the variables that can be read and the other is all variables that can be assigned new values (therefore read/write variables appear in both sets). For example, this may include loop counters that should only be changed by the loop statement, or parameters passed to a function *e.g.* a list of integers to be summed. If the list were changed during the execution then the sum might not be accurate for the given list. As well as separating variables by access permissions, it is also possible to separate by type. For example, a list of integers and a list of floats can be kept separate and the statements designed to preserve type correctness.

One final extension that is worth mentioning is the use of a counter to limit the running time of the code. This only needs to be incremented each iteration of a loop and can be used to terminate execution of excessively long programs. For the examples presented here, the Perl 'eval' function is used to execute the evolved programs and this sets a variable with an error message when there is an unnatural termination of the execution. This can be used to detect errors and also for limiting the execution time of a program.

3.5. Wrapper

It will usually be convenient to add some header and footer code to the evolved code for the purposes of declaring variables, receiving data

Table I. Example Genome

28	34	64	124
127	130	33	83
201	5	41	50
201	9	69	73

Table II. Statement type, type of additional genes, and form of statement

Index	Statement	Additional Genes	Format
0	Assign	variable,variable	`G1 = G2;`
1	Multiply	variable,variable,variable	`G1 = G2 * G3;`
2	If	variable,comparison,variable	`if(G1 G2 G3){`
3	For	variable,variable	`for G1 (0..G2){`
4	End		}

passed to the evolved code and returning data after the code has been executed. This could be included as part of the evolution process but would make the problem much harder without real benefit, for example see Listing 1. This extra code is used in the experiments to allow the use of the Perl 'eval' function to test the evolved programs.

3.6. Example

As an example of the mapping from the genome to the phenome using the above method, a function to calculate the factorial of a number is presented.

Table I shows the genome (the list of integers). This genome is converted using the statements listed in Table II and the additional genes are translated using Table III. As can be seen from Table II, the most additional genes required by a statement is three. Therefore, the length of each gene block will be four (to include the choice of statement).

Table III. List of variables and comparison operators

Index	Variable
0	`$n`
1	`$fact`
2	`$count`
3	`$zero`

Index	Comparison
0	`==`
1	`!=`
2	`>`
3	`<`

Table IV. Conversion of Genome to Phenome

Genes	Index	Statement	Code
28,34,64,124	3,2,0,x	For	`for $count (0..$n){`
127,130,33,83	2,2,1,3	If	`if($count != $zero){`
201,5,41,50	1,1,1,2	Multiply	`$fact = $fact * $count;`
94,231,0,13	4,x,x,x	End	}

Listing 1: The entire phenome, including header and footer

```
# Header
my $n = $ARGV[0];
my $fact = 1;
my $count = 0;
my $zero = 0;

# Evolved Code
for $count (0..$n){
 if($n != $zero){
  $fact = $fact * $count;
 }
}

# Footer
return $fact;
```

The first block of genes starts with the value 28. This represents the statement type being used. In this case there are five types of statements, so $28 \equiv 3 (\mathrm{mod}\ 5)$ means that the statement is a 'For' (index 3). The 'For' statement requires the use of two more genes to choose from the 'variable' list. The 'variable' list has four elements, therefore, $34 \equiv 2 (\mathrm{mod}\ 4)$ and $64 \equiv 0 (\mathrm{mod}\ 4)$ give the variables `$count` and `$n`. All together this gives the loop header `for $count (0..$n){`. The gene 124 is redundant. The rest of the gene blocks are decoded in the same way (see Table IV).

Finally, any missing closing braces are automatically added, along with the wrapper (header and footer) code, to create the complete phenome. This is shown in Listing 1.

Table V. List of statements for padding test

Index	Statement	Additional Genes	Format
0	Print	variable	`print G1;`
1	For	variable,variable	`for G1 (0..G2)`
2	Add	variable,variable,variable	`G1 = G2 + G3;`

Table VI. List of variables for padding test

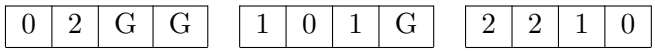
Index	Variable
0	`$x`
1	`$y`
2	`$z`

4. Comparison of Padded and Unpadded Representations

This section compares the fixed-length gene blocks (padded with redundant genes) with variable-length gene blocks (unpadded – as used in *e.g.* Grammatical Evolution [37]), to examine the preservation of characteristics after mutation and crossover with another individual. The simple set of statements listed in Table V and the set of variables listed in Table VI are used to map the genomes to the phenomes with the method presented in Section 3.3.

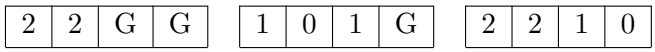
The first experiment is to compare how the padded version of an individual changes under mutation in comparison with an unpadded individual. Figure 3a shows an example individual with fixed-length gene blocks representing the statements and Figure 4a shows the same individual without the redundant genes. The 'G' represents an unused gene in the padded genome, although these genes may be used after mutation or crossover and are shown in the phenomes as 'G' when used. The mapping for the unpadded individual's genome to phenome just uses the relevant number of genes for each statement and starts the next statement immediately afterwards.

Figure 3b shows the first individual (Figure 3a) after a mutation of the first gene (from 0 to 2). It can be seen that only the first statement of the phenome has changed and the rest is identical to the pre-mutation version. In contrast, Figure 4b shows the unpadded individual (Figure 4a) after the same mutation. The phenome of the individual is now completely different, very little has been preserved from the original individual. This would not be good from the perspective of the evolution as good characteristics, which caused the individual to




```
print $z;
for $x (0..$y) {
 $z = $y + $x;
}
```

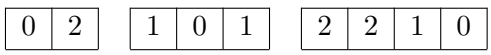
(a)




```
$z = G + G;
for $x (0..$y) {
 $z = $y + $x;
}
```

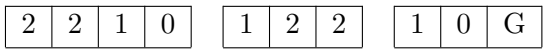
(b)

Figure 3. (a) Parent 1 (Padded), (b) Parent 1 (Padded) Mutated




```
print $z;
for $x (0..$y) {
 $z = $y + $x;
}
```

(a)



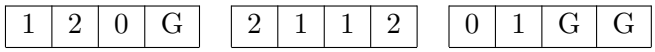

```
$z = $y + $x;
for $z (0..$z) {
 for $y (0..G) {
 }
}
```

(b)

Figure 4. (a) Parent 1 (Unpadded), (b) Parent 2 (Unpadded) Mutated

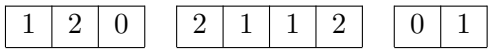
be selected for reproduction, are lost whereas with the padded version most the characteristics are preserved.

This problem would be expected to be even more pronounced when using crossover, as there is much more change when the individuals are combined. Figures 5a and 5b show the padded and unpadded versions of a second individual, which both map to the same phenome. When




```
for $z (0..$x) {
 $y = $y + $z;
 print $y;
}
```

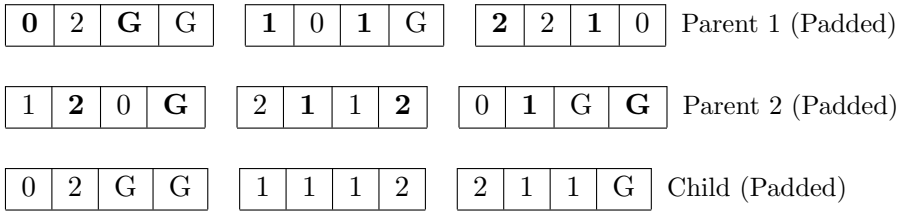
(a)




```
for $z (0..$x) {
 $y = $y + $z;
 print $y;
}
```

(b)

Figure 5. (a) Parent 2 (Padded), (b) Parent 2 (Unpadded)




```
print $z;
for $y (0..$y) {
 $y = $y + $x;
}
```

Figure 6. Crossover Parent 1 and Parent 2 (Padded)

the padded versions of Parent 1 (Figure 3a) and Parent 2 (Figure 5a) are combined using crossover (taking alternate genes starting with the first individual in this case) the individual in Figure 6 is created. This individual looks quite similar to the first parent, as the main statement type gene is always taken from this individual (in this example) because the gene-length is even. When the unpadded individuals (Figures 4a and 5b) are combined in the same way as the padded individuals, Figure 7 is produced. Apart from maintaining the first statement type of the first individual, it is completely different to either parent. The final gene 'G' in Figure 7 represents an extra gene required. The alternative is to not use any gene block with insufficient genes, however, this is not an issue with the padded versions.

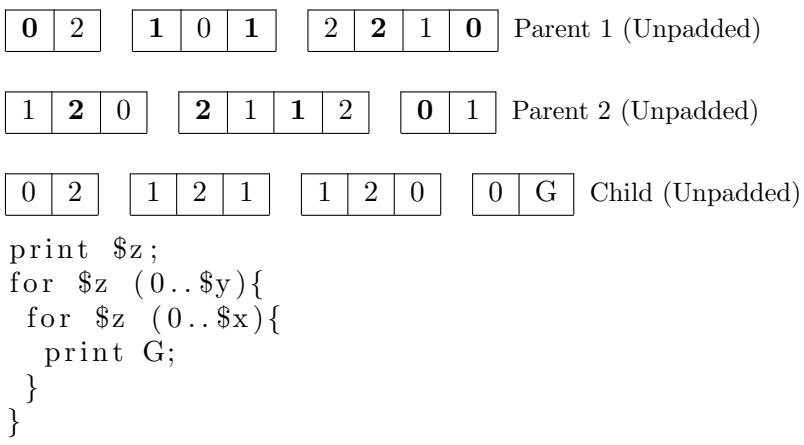


Figure 7. Crossover Parent1 and Parent2 (Unpadded)

In conclusion, the small examples shown suggest that inheritable characteristics are much more likely to be preserved when using fixed-length gene blocks to represent individual statements. However, mutation and crossover can still give variation to the child individuals without losing similarity to the parents.

5. Experimental Setup

In this section, the previously presented representation is applied to the evolution of a series of list evaluation and manipulation functions. The particular functions being evolved have some interesting features, which set them apart from traditional GP problems. Firstly, the functions need to achieve a 100% fitness level to be useful. This moves away from the traditional GP optimization problems, where the idea is to improve upon current results and just requires search of the solution space for adequately fit individuals. Secondly, these functions are commonly used as part of larger programs, so can be used as components to create larger programs after they have been evolved (for example see [46]). Yu [47] also used two simple list functions for testing; finding the nth element of a list and applying a predefined function to a list. More complex programs have been evolved by Genetic Algorithms *e.g* Groß' chess playing program [14], however, this lacked to 100% fitness requirement.

The following functions were evolved:

Sumlist — find the sum of a list of integers.

Avelist — find the average of a list of integers.

Listmax — find the largest value in a list of integers.

Listmin — find the smallest value in a list of integers.

Reverse — reverse the ordering of a list of integers.

Sort — sort a list of integers into ascending order.

This set of problems starts with two functions that return numerical results (that do not necessarily appear in the input lists), the second two functions return elements of the input lists and the final two return new lists. This may be thought of as a series of problems of apparent increasing difficulty.

5.1. The Underlying Genetic Algorithm

The following is a description of the simple Genetic Algorithm which is used for all of the experiments. The algorithm is kept simple to keep the focus on the effects of the representation and the fitness functions.

Algorithm 1 The simple Genetic Algorithm used for all experiments

```
P = Initialise_Population
F = Test_Fitness(P)
for generations = 1 to MAXGEN do
  P = Reproduce(P, F)
  F = Test_Fitness(P)
end for
```

The simple Genetic Algorithm used is given in Algorithm 1. The genomes are represented as fixed length integer strings, as this is the easiest representation to work with in terms of the genome/phenome mapping and the genetic manipulation. The representation does not, however, preclude the use of variable length genomes. In the reproduction function, simple fitness proportionate parent selection is used to select two parents, these two parents are combined using uniform crossover and then there is a probability that each gene will be mutated in the resulting individual. The best individual from the previous generation is copied into the newly created population.

In the experiments, a relatively small population of 7 and a relatively high mutation rate of 10% are used. These values are much smaller than are traditionally used, more like an Evolution Strategy than a Genetic Algorithm, however, they appear to produce high fitness individuals quickly.

The GA is run for a maximum of $50,000$ generations with 50 different random seeds for each problem. In addition, the execution time of each individual is limited by keeping count of the number of iterations of loops. If the counter reaches 1000 the program is terminated with an error and receives a minimal fitness value.

Finally, the experiments were run on a 3.4GHz Xeon (1MB cache) PC with 3GB of RAM, running the Ubuntu Server GNU/Linux (Dapper) 6.06 operating system. The programs were written in Perl 5.8.7, as was the evolved code.

5.2. Fitness Testing

One way a fitness function for Genetic Programming can be constructed is to use a set of sample inputs for a problem and compare the resultant outputs from the evolved function with the expected outputs. The fitness function can also be some 'hand-crafted' evaluation function, in which case it is difficult to guarantee that all features of the problem have been covered, especially for larger problems. While the first method may be suitable for some simple situations, it is unlikely to generate an accurate fitness score, on more complex problems, without a very large number of test inputs. Cramer [6] mentions that a major problem in evolving programs is one of 'hand-crafting' the evaluation function to give partial credit to a function that does not work but exhibits some of the relevant behaviour. In [24], for example, Koza hand-crafts his fitness functions based on the natural terminology of the problem but gives little justification of his choices. In these experiments fitness evaluation functions based on the formal specification of the problems are used in conjunction with input/output pairs. This removes the need to write *ad hoc* functions to test fitness and replaces it with a disciplined alternative. See, for example, the *Listmax* problem (specified in Figure 10). Using fitness measured by counting the number of correct solutions from given test inputs a fitness of zero would be given for a function returning the second highest value in the list, whereas the fitness function created from the formal specification would return quite a high fitness value.

Figures 8 to 13 and Listings 2 to 7 give the formal specifications and conversion into code for the experiments. For example, the specification of *sumlist* (Figure 8) simply says "the output value s is equal to the sum of the list L". The notation $(+x|x : \mathbb{Z}, L_1, L_2 : \mathbb{Z}^*|L = L_1{}^\frown\langle x\rangle{}^\frown L_2)$, used in the specification, simply means "take each element x in the list L and add it to the other elements in the list". The notation used is described in more detail in [4]. The conversion to code is shown in Listing 2. One adjustment to the natural conversion of giving one fitness

$$sumlist : \mathbb{Z}^* \rightarrow \mathbb{Z}$$
$$\text{pre-}sumlist(L) \triangleq \#L \neq 0$$
$$\text{post-}sumlist(L, s) \triangleq s = (+x | x : \mathbb{Z}, L_1, L_2 : \mathbb{Z}^* | L = L_1 \frown \langle x \rangle \frown L_2)$$

Figure 8. Specification of *sumlist*

Listing 2: The fitness function for *sumlist* (without header)

```
foreach my $x (@L){
 $t += $x;
}
$fitness += abs($s-$t);
```

point for each correct element of the specification, is to use the absolute difference between the expected and actual results. This adjustment is due to this particular specification only having one test and therefore a very limited hill to climb. However, with the change the hill becomes much bigger and less steep. Initial tests showed that the *avelist* function found great difficulty evolving without the change. The use of absolute differences in values was not used in the remaining four problems, as there were more tests.

5.3. Language Subset

Tables VII, VIII and IX give the list of statements and the meaning of additional genes for the *sumlist* and *avelist* experiments. For the *avelist* experiment, the variable `$sum` is replaced by the variable `$ave`. The set of statements used is fairly constrained. The statements differentiate between variables that are read-only and those that can be assigned new values. Variables, such as list counters, that are automatically updated are considered read-only, so that they, for example, cannot be changed within the loop body. One final interesting feature is the list index (Index 3 of the read-only variables, *rvars* in Table IX), which is constrained to only be able to reference elements within the list by

$$avelist : \mathbb{Z}^* \rightarrow \mathbb{R}$$
$$\text{pre-}avelist(L) \triangleq \#L \neq 0$$
$$\text{post-}avelist(L, s) \triangleq s = (+x | x : \mathbb{Z}, L_1, L_2 : \mathbb{Z}^* | L = L_1 \frown \langle x \rangle \frown L_2)/\#L$$

Figure 9. Specification of *avelist*

Listing 3: The fitness function for *avelist* (without header)

```
foreach my $x (@L){
 $t += $x;
}
$fitness += abs($s-($t/($#L+1)));
```

$$
\begin{aligned}
&\mathit{listmax} : \mathbb{Z}^* \to \mathbb{Z} \\
&\text{pre-}\mathit{listmax}(L) \triangleq \#L \neq 0 \\
&\text{post-}\mathit{listmax}(L, m) \triangleq m \text{ in } L \wedge (\forall z : \mathbb{Z})(z \text{ in } L \Rightarrow z \leq m) \\
&\qquad \text{where } (x \text{ in } L) \triangleq (\exists L_1, L_2 : \mathbb{Z}^*)(L = L_1 \frown \langle x \rangle \frown L_2)
\end{aligned}
$$

Figure 10. Specification of *listmax*

Listing 4: The fitness function for *listmax* (without header)

```
foreach my $z (@L){
 if($z <= $m){$fitness++;}
}
```

$$
\begin{aligned}
&\mathit{listmin} : \mathbb{Z}^* \to \mathbb{Z} \\
&\text{pre-}\mathit{listmin}(L) \triangleq \#L \neq 0 \\
&\text{post-}\mathit{listmin}(L, m) \triangleq m \text{ in } L \wedge (\forall z : \mathbb{Z})(z \text{ in } L \Rightarrow z \geq m) \\
&\qquad \text{where } (x \text{ in } L) \triangleq (\exists L_1, L_2 : \mathbb{Z}^*)(L = L_1 \frown \langle x \rangle \frown L_2)
\end{aligned}
$$

Figure 11. Specification of *listmin*

Listing 5: The fitness function for *listmin* (without header)

```
foreach my $z (@L){
 $fitness++ if($z >= $m);
}
```

Listing 6: The fitness function for *reverse* (without header)

```
for(my $n=0; $n<scalar(@L); $n++){
 if($L[$n] == $N[$#L-$n]){$fitness++;}
}
```

$$
\begin{aligned}
&reverse : \mathbb{Z}^* \rightarrow \mathbb{Z}^* \\
&\text{pre-}reverse(L) \triangleq True \\
&\text{post-}reverse(L, N) \triangleq \#L = \#N \\
&\quad \wedge (\forall n : \mathbb{P})(n < \#L \Rightarrow \\
&\qquad (\exists x : \mathbb{Z}, L_1, L_2, N_1, N_2 : \mathbb{Z}^*)(\quad L = L_1 \frown \langle x \rangle \frown L_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge N = N_1 \frown \langle x \rangle \frown N_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \#L_1 = \#N_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \#L_1 = n))
\end{aligned}
$$

Figure 12. Specification of *reverse*

$$
\begin{aligned}
&sort : \mathbb{Z}^* \rightarrow \mathbb{Z}^* \\
&\text{pre-}sort(L) \triangleq True \\
&\text{post-}sort(L, N) \triangleq bag\_of(N) = bag\_of(L) \wedge ascending(N) \\
&\quad \text{where} \quad bag\_of(\langle\rangle) \blacktriangleleft\blacktriangleright \emptyset \\
&\qquad\qquad\quad bag\_of(\langle x \rangle) \blacktriangleleft\blacktriangleright \{|x|\} \\
&\qquad\qquad\quad bag\_of(L_1 \frown L_2) \blacktriangleleft\blacktriangleright bag\_of(L_1) \uplus bag\_of(L_2) \\
&\quad ascending(N) \triangleq (\forall x, y : \mathbb{Z})(x \text{ before } y \text{ in } N \Rightarrow x \leq y) \\
&\text{and} \\
&\quad x \text{ before } y \text{ in } N \triangleq (\exists N_1, N_2, N_3 : \mathbb{Z}^*)(N = N_1 \frown \langle x \rangle \frown N_2 \frown \langle y \rangle \frown N_3)
\end{aligned}
$$

Figure 13. Specification of *sort*

taking the modulus of the size of the list. This seemed to be a fairly logical way of constraining the list index.

Table X and XI show the list of statements and meaning of additional genes for the *listmax* and *listmin* problems. As the functions only return a value from the list of integers no arithmetic statements are included in the statement list. As with *sumlist* and *avelist* the list

Listing 7: The fitness function for *sort* (without header)

```
if($#N > 0){
 for my $x (0..$#N-1){
  $fitness++ if($N[$x] <= $N[($x+1)]);
 }
}
```

Table VII. List of statements used for *sumlist*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	Add	wvars,rvars,rvars	`G1 = G2 + G3;`
3	Subtract	wvars,rvars,rvars	`G1 = G2 - G3;`
4	Multiply	wvars,rvars,rvars	`G1 = G2 * G3;`
5	Divide	wvars,rvars,rvars	`G1 = G2 / G3 if(G3 != 0);`
6	If	rvars,cmp,rvars	`if(G1 G2 G3){`
7	For	rvars,lsize	`for G1 (0..G2){`
8	End		}

Table VIII. List of statements used for *avelist*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	Add	wvars,rvars,rvars	`G1 = G2 + G3;`
3	Divide	wvars,rvars,rvars	`G1 = G2 / G3 if(G3 != 0);`
4	For	rvars,lsize	`for G1 (0..G2){`
5	End		}

Table IX. Additional Genes for *sumlist* and *avelist*

Index	wvars	rvars	lsize	cmp
0	`$sum`	`$sum`	`$#list`	`==`
1		`$size`		`!=`
2		`$tmp`		`>`
3		`$list[$tmp%($#list+1)]`		`<`
4				`>=`
5				`<=`

Table X. List of statements used for *listmax* and *listmin*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	If	rvars,cmp,rvars	`if(G1 G2 G3){`
3	For	rvars,lsize	`for G1 (0..G2){`
4	End		}

Table XI. Additional Genes for *listmax* and *listmin*

Index	wvars	rvars	lsize	cmp
0	`$max`	`$max`	`$#list`	`==`
1		`$tmp1`		`!=`
2		`$tmp2`		`>`
3		`$list[$tmp1%($#list+1)]`		`<`
4		`$list[$tmp2%($#list+1)]`		`>=`
5				`<=`

indexes are constrained to only index valid list elements. For *listmin* the variable `$max` is replaced with `$min`.

Tables XII and XIII give the list of statements used and the meaning of additional genes for the *reverse* problem. It is interesting to note the provision of list indexing of the form `L[#L-n]`. This is a logical choice based on the interpretation of the specification.

Tables XIV and XV give the list of statements used and the meaning of additional genes for the *sort* problem. The list of statements now has the additional statement type the 'Double' loop (Index 4). This is a standard structure used when comparing elements in a list. In addition, it also demonstrates how larger building blocks can be used to evolve programs (other possible building blocks for this problem could include the 'Swap' function, commonly used in sort algorithms).

5.4. Test Inputs

The list of test inputs used for fitness testing with *sumlist*, *avelist*, *reverse* and *sort* is given in Listing 8. The test set includes a variety of different length lists and a variety of orderings of the elements with the

Table XII. List of statements used for *reverse*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	`G1 = G2;`
2	Add	wvars,rvars,rvars	`G1 = G2 + G3;`
3	Subtract	wvars,rvars,rvars	`G1 = G2 - G3;`
4	Multiply	wvars,rvars,rvars	`G1 = G2 * G3;`
5	Divide	wvars,rvars,rvars	`G1 = G2 / G3 if(G3 != 0);`
6	If	rvars,cmp,rvars	`if(G1 G2 G3){`
7	For	rvars,lsize	`for G1 (0..G2){`
8	End		}

Table XIII. Additional Genes for *reverse*

Index	wvars	rvars	lsize	cmp
0	`$out[$tmp1]`	`$tmp1`	`$#in`	`==`
1	`$out[$#in - $tmp1]`	`$tmp2`		`!=`
2	`$out[$tmp2]`	`$in[$tmp1]`		`>`
3	`$out[$#in - $tmp2]`	`$in[$#in - $tmp1]`		`<`
4		`$in[$tmp2]`		`>=`
5		`$in[$#in - $tmp2]`		`<=`
6		`$out[$tmp1]`		
7		`$out[$#in - $tmp1]`		
8		`$out[$tmp2]`		
9		`$out[$#in - $tmp2]`		

Table XIV. List of statements used for *sort*

Index	Statement	Additional Genes	Form
0	Null		
1	Assign	wvars,rvars	G1 = G2;
2	If	rvars,cmp,rvars	if(G1 G2 G3){
3	For	counter,lsize	for G1 (0..G2){
4	Double	counter,lsize,counter	for G1 (0..G2){ for G3 (G1+1..G2){
5	End		}

Table XV. Additional Genes for *sort*

Index	wvars	rvars	counter	lsize	cmp
0	`$in[$tmp1]`	`$in[$tmp1]`	`$tmp1`	`$#in`	`==`
1	`$in[$tmp2]`	`$in[$tmp2]`	`$tmp2`		`!=`
2	`$tmp3`	`$tmp1`			`>`
3	`$tmp4`	`$tmp2`			`<`
4		`$tmp3`			`>=`
5		`$tmp4`			`<=`

Listing 8: Set of test input lists for *sumlist*, *avelist*, *reverse* and *sort*

```
[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]
```

lists *i.e.* the largest and smallest elements are not always in the same locations.

The set of test inputs for *listmax* and *listmin* is given in Listing 9. The test set includes lists of varying lengths with both positive and negative integers. The number of tests used can affect the performance of the GP from both the perspective of overall time to evaluate fitness and the number of generations required to evolve a fully fit solution. The results, in this case, show that the number of test cases was sufficient.

Listing 9: Set of test input lists for *listmax* and *listmin*

```
[1,4,2,32,345],
[-42,-34,-12,-235],
[46,0,2,23],
[54,13,1,24,235,35],
[12,245,6]
```

Listing 10: Set of verification input lists for all tests

```
[-52,15],
[-36,59,49,-3],
[29],
[24,-1,60,-72,-63],
[-43,54,-11,-16,56],
[45,17,82,58,28,84,21,67,-98],
[13,-52,47,-34],
[32,16,-64,-11,-53,-32,45,61,-36],
[76,20,-44,8],
[14,-88,-20,51]
```

Finally, Listing 10 shows the list of verification inputs used to check solutions that have achieved maximum performance with the test inputs.

6. Results

This section presents the results of the experiments to evolve the various list evaluation and manipulation functions. For consistency the first run (seed 0) is always used as the example solution. All examples are shown with their wrapper code included. Table XVI presents a summary of the results for the fifty runs of each experiment. Table XVII shows the number of runs that gained 100% fitness on the verification test set.

For *sumlist*, all the runs evolve a fully fit individual within a very small number of generations (and a very short time). Even though all the results are achieved very quickly, there is still some considerable variation in the values. This is the expected result due to the non-deterministic nature of GA and GP. Only two of the evolved solutions failed to achieve 100% fitness on the verification set.

Listing 11 shows an example individual evolved. Taking into account that the first two lines of code are redundant, as is the nested 'If' statement and its contents, the resultant program is the same as might be written by a 'real' programmer and will, therefore, find the sum of any input list.

For *avelist*, the results are typically slightly slower than those of the *sumlist* function (although the mean runtime is significantly larger due to two of the runs failing to find an optimal solution). All the other runs produced a fully fit individual in a fast time and few generations.

Table XVI. Results summary from 50 runs of each experiment

		Mean	S.D.	Median	Min	Max
Sumlist	Runtime (s)	1.35	1.09	1.04	0.07	5.55
	Generations	121.08	96.04	94.5	6	493
Avelist	Runtime (s)	63.06	131.55	9	0.45	639.17
	Generations	5198.64	10355.44	816.5	43	50000
Listmax	Runtime (s)	0.87	0.66	0.63	0.1	4
	Generations	129.12	95.68	94	16	576
Listmin	Runtime (s)	2.23	2.91	1.29	0.13	17.34
	Generations	334	430.59	199	19	2552
Reverse	Runtime (s)	1.84	1.6	1.4	0.14	8.81
	Generations	116.76	102.45	91.5	9	570
Sort	Runtime (s)	361	270.27	298.03	30.96	947.1
	Generations	19137.74	14282.48	16002.5	1661	50000

Table XVII. Number with 100% verification score (out of 50)

Sumlist	Avelist	Listmax	Listmin	Reverse	Sort
48	48	28	47	44	22

Listing 12 shows an example of a generated fully fit individual. The first three lines of code are redundant to the functionality of the program and the remaining code is as would be expected (assuming a valid input to the function) if the function was 'hand-coded'.

For *listmax*, all runs produce a fully fit individual in a very short time. The performance is similar to that of *sumlist* and *avelist*. 28 out of the 50 runs resulted in a solution that achieved a 100% score on the verification set.

Listing 13 gives an example fully fit individual evolved by the GP. This particular example works with the test input set and other similar lists but not in the general case. For example, the list `[-1,3,2,1]` returns the value 1. However, some of the functions evolved do return the correct value in the general case (this was determined by inspection of the functions).

For *listmin*, the number of generations and running times are slightly higher than the *listmax* experiment but not significantly. 47 of the runs achieved a 100% score on the verification set.

An example fully fit program evolved is given in Listing 14. As with *listmax*, this example does not completely solve the problem but

Listing 11: Example solution for *sumlist*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $sum = 0;
my $size = $#list+1;
my $tmp = 0;

# Evolved Code
$sum = $size;
$sum = $tmp;
for $tmp (0..$#list){
  $sum = $sum + $list[$tmp%($#list+1)];
  if($list[$tmp%($#list+1)] <
     $list[$tmp%($#list+1)]){
    for $tmp (0..$#list){
    }
  }
}

# Footer
return $sum;
```

Listing 12: Example solution for *avelist*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $ave = 0;
my $size = $#list+1;
my $tmp = 0;

# Evolved Code
$ave = $tmp + $ave;
for $tmp (0..$#list){
}
for $tmp (0..$#list){
  $ave = $list[$tmp%($#list+1)] + $ave;
}
$ave = $ave / $size if($size != 0);

# Footer
return $ave;
```

Listing 13: Example solution for *listmax*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $max = 0;
my $tmp1 = 0;
my $tmp2 = 0;

# Evolved Code
for $tmp2 (0..$#list){
  if($list[$tmp1%($#list+1)] <=
     $list[$tmp2%($#list+1)]){
    $max = $list[$tmp2%($#list+1)];
    if($tmp2 >= $max){
      if($list[$tmp1%($#list+1)] == $max){
        $max = $list[$tmp2%($#list+1)];
        for $tmp1 (0..$#list){
        }
      }
    }
  }
}

# Footer
return $max;
```

does work with all of the test inputs. Some of the solutions not only satisfy the test data but can be manually proven to conform to the specification.

For *reverse*, the runs all evolve fully fit individuals very quickly (both in terms of time and generations). This, slightly unexpected, performance is most likely due to the inclusion of the `L[#L-n]` variables, which would appear to make the problem much easier. 44 of the runs produced a program that achieved a 100% score on the verification set.

Listing 15 gives an example individual evolved. When the redundant parts are removed the function appears similar to the expected general solution, with the inclusion of a few additional statements that do not affect the functionality.

For the *sort* experiment, it is clear that this problem is much harder for the GP to solve. The times and number of generations required to evolve a fully fit individual are considerably higher than the previous experiments. However, all of the runs still produce a fully fit individual

Listing 14: Example solution for *listmin*, Seed 0

```
# Header
my @list = @{$test[$t]};
my $min = 0;
my $tmp1 = 0;
my $tmp2 = 0;

# Evolved Code
for $tmp2 (0..$#list){
  if($tmp2 != $list[$tmp1%($#list+1)]){
    $min = $list[$tmp1%($#list+1)];
    for $tmp2 (0..$#list){
      if($min > $list[$tmp2%($#list+1)]){
        $min = $list[$tmp2%($#list+1)];
      }
    }
  }
}

# Footer
return $min;
```

within the number of generations allowed. The increased difficulty is possibly due to there being a small number of *correct* solutions in the set of all possible genomes. In addition, there could be local minima around the optimal solutions, which would make it difficult for the GP to produce perfect individuals quickly. Only 22 of the runs produced a 100% fitness score on the verification set.

Listing 16 shows an example solution from the *sort* experiment. This is nearly the same as a 'bubble' sort, however, the inclusion of the two additional 'If' statements in the body of the 'Double' loop means that the function will not work in all cases. Some of the runs did evolve solutions that work in the general case. All of the solutions are variations on the 'bubble' sort. This is most likely due to the constraints of the language subset used. However, it is interesting to note that not all of the solutions took advantage of the 'Double' statement.

6.1. Summary of results

To summarise, a set of list evaluation and manipulation functions were evolved, with the interesting feature that they needed to be completely

Listing 15: Example solution for *reverse*, Seed 0. All of the list indices are taken modulo the size of the list, however this code is not shown for clarity

```
# Header
my @in = @{$test[$t]};
my @out = ();
my $tmp1 = 0;
my $tmp2 = 0;

# Evolved Code
$out[($#in - $tmp2)] = $out[($#in - $tmp2)];
for $tmp1 (0..$#in){
  if($in[($#in - $tmp2)] < $in[$tmp1]){
  }
  $out[$tmp1] = $tmp2 + $in[($#in - $tmp1)];
  if($in[$tmp2] <= $in[$tmp2]){
    $out[$tmp1] = $out[($#in - $tmp1)] /
      $in[$tmp2] if($in[$tmp2] != 0);
    $out[$tmp1] = $in[($#in - $tmp1)];
  }
}

# Footer
return @out;
```

correct to be useful. The experiment showed that it was possible to evolve these functions, with this constraint, in a reasonably short amount of time. The *sort* function, however, showed that as the complexity of the problem rises, the time to solve the problem also rises quite sharply. This suggests that more complex problems would be nearly impossible to evolve as one block of code. There are two approaches to combating this problem. Either the larger functions can be evolved from smaller blocks of code and calls to previously evolved functions or the problem can be broken down into a series of smaller problems that can be evolved, possibly in parallel. This is addressed in Section7.

7. Addressing Scalability

One approach to scalability is to define the requirements for the function in a hierarchical way, so that the function can be evolved from

Listing 16: Example solution for *sort*, Seed 0. All of the list indices are taken modulo the size of the list, however this is not shown in the code for clarity

```
# Header
my @in = @{$test[$t]};
my $tmp1 = 0;
my $tmp2 = 0;
my $tmp3 = 0;
my $tmp4 = 0;

# Evolved Code
for $tmp2 (0..$#in){
  for $tmp1 ($tmp2+1..$#in){
    if($in[$tmp2] > $in[$tmp1]){
      if($in[$tmp1] != $tmp2){
        $tmp3 = $in[$tmp1];
      }
      if($tmp3 <= $in[$tmp2]){
        $in[$tmp1] = $in[$tmp2];
      }
      $in[$tmp2] = $tmp3;
    }
  }
}

# Footer
return @in;
```

sub-functions. This is relatively easy to do with a formal specification of the function. For example, the post-condition for *avelist* might look like

$$\text{post-}avelist(L, s) \triangleq s = sumlist(L)/\#L$$

and the *sumlist* function would be added to the language subset available. In addition, compound statements can be used such as the 'Double' statement in the previous experiments.

7.1. Sort with 'Swap' Function

As an example of using functions and compound statements to improve the performance of the evolution process, the *sort* example from the previous experiments, is extended. The GP was already using the com-

Table XVIII. Results summary from 50 runs of sort with swap experiment

		Mean	S.D.	Median	Min	Max
Sort (with swap)	Runtime	6.9	5.4	5.95	0.22	25.26
	Generations	334.74	258.36	296	11	1270

Listing 17: Example of the *sort* function using 'Swap', Seed 1

```
if($inlist[$tmp2] >= $tmp3){
  for $tmp2 (0..$#inlist){
    for $tmp1 ($tmp2+1..$#inlist){
      if($inlist[$tmp2] > $inlist[$tmp1]){
        swap($inlist[$tmp1],$inlist[$tmp2]);
      }
    }
  }
}
```

pound 'Double' statement. To the previously used set of statements is added the function 'Swap', which is common to many sort functions, and swaps two elements of a list. The rest of the language subset and test input is left unchanged (see Tables XIV, XV and Listing 8). The fitness function also remains unchanged (see Listing 7).

Table XVIII shows a summary of the results of the 50 runs of the experiment. It can be clearly seen that the introduction of the 'Swap' function had a dramatic impact on the performance of the GP. From an average number of generations of 19138, and an average time of 361s in the original experiment, the average number of generations is now just 335 and the average time is just 7s. This is an improvement of roughly two orders of magnitude. In addition, 24 out of 50 passed verification.

Listing 17 shows an example function generated. Apart from the surrounding 'If' statement, the code is the expected 'bubble' sort using the 'Swap' function to exchange elements of the list.

8. Discussion and Conclusions

A representation for programs, along with a mapping between the genome and phenome, has been presented. The representation has the benefits of explicitly inheritable characteristics, easy mapping between the genome and phenome, support for arbitrary genetic operators, and

the ability to represent programs in any language. This representation was shown to have better inheritance characteristics between individuals than systems such as Grammatical Evolution [37]. In addition, all genomes map to a complete program without reusing genes or randomly extending the genome.

A series of functions was evolved that were more traditional programming problems than traditional GP problems. The functions had the need for a 100% fitness value over the given set of test inputs to be considered useful, although even when this condition is met it is difficult to guarantee that the function matches the specification for all inputs. However, this problem is not unique to the evolution of functions, when humans write computer programs the same problem exists. Previous researchers (*e.g.* Koza [24]) have applied GPs to problems where the solutions have a *better/worse* classification (such as 'Traveling Salesman') rather a *right/wrong* classification. The experiments presented in this paper have shown that GPs can be applied to a wider selection of programming problems.

A method of dealing with scalability issues was briefly introduced, which evolved larger functions from more abstract code segments such as compound statements and function calls. This showed a dramatic improvement from the previously presented version of the experiment.

One of the biggest problems highlighted by the experiments is that an individual gaining a 100% fitness value isn't always correct in the general case. This is due to the test input set not being exhaustive (if it were, the time to fitness test an individual would be impractical). In addition to the dramatic improvement in the time to evolve the sort program from basic building blocks the addition of the compound statement as a basic element resulted in a significant increase in the number of solutions that could be validated.

References

1. Angeline, P. J. and J. B. Pollack: 1992, 'The evolutionary induction of subroutines'. In: J. Kruschke (ed.): *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum, Hillsdale, NJ, USA.
2. Banzhaf, W.: 1993, 'Genetic Programming for Pedestrians'. In: S. Forrest (ed.): *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann, University of Illinois at Urbana-Champaign, p. 628.
3. Banzhaf, W., P. Nordin, R. Keller, and F. Francone: 1998, *Genetic Programming — An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann.
4. Cooke, D.: 1998, *Constructing Correct Software: the basics*. London, UK: Springer-Verlag.

5. Cooper, J. and C. Hinde: 2003, 'Improving genetic algorithms' efficiency using intelligent fitness functions'. In: P. Chung, C. Hinde, and M. Ali (eds.): *16th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE '03, Loughborough, UK, June 23-26, 2003, Proceedings*. Springer, Berlin, pp. 636–644.
6. Cramer, N.: 1985, 'A Representation for the Adaptive Generation of Simple Sequential Programs'. In: J. Grefenstette (ed.): *Proceedings of the First International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, pp. 183–187.
7. Fogel, D.: 1998, *The Fossil Record*. IEEE Press.
8. Fogel, L., A. Owens, and M. Walsh: 1966, *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, Inc.
9. Freeman, J. J.: 1998, 'A linear representation for GP using context free grammars'. In: J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, D. Fogel, M. Garzon, D. Goldberg, H. Iba, and R. Riolo (eds.): *Genetic Programming 1998: Proceedings of the third annual conference*. Morgan Kaufmann, San Francisco, CA, pp. 72–77.
10. Friedberg, R.: 1958, 'A Learning Machine: Part I.'. *IBM J. Research and Development* **2**(1), 2–13.
11. Friedberg, R., D. B., and T. North: 1959, 'A Learning Machine: Part II.'. *IBM J. Research and Development* **3**(3), 282–287.
12. Fujiki, C.: 1986, 'An Evaluation of Holland's Genetic Operators Applied to a Program Generator'. Master's thesis, University of Idaho, Moscow, ID.
13. Fujiki, C. and J. Dickinson: 1987, 'Using the Genetic Algorithm to Generate LISP Source Code to Solve the Prisoner's Dilemma'. In: J. Grefenstette (ed.): *Genetic Algorithms and their Applications: Proc. of the 2nd Intern. Conf. on Genetic Algorithms*. Lawrence Erlbaum, pp. 236–240.
14. Gross, R., K. Albrecht, W. Kantschik, and W. Banzhaf: 2002, 'Evolving Chess Playing Programs'. In: W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (eds.): *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, pp. 740–747.
15. Harrison, M.: 1978, *Introduction to Formal Language Theory*. London, UK: Addison Wesley.
16. Hemberg, M., U. M. O'Reilly, and P. Nordin: 2001, 'GENR8: A Design Tool for Surface Generation'. In: H. Beyer, E. Cantu-Paz, D. Goldberg, S. Parmee, and D. Whitley (eds.): *Late Breaking Papers, GECCO 2001*. Morgan Kaufmann.
17. Hinklin, J.: 1986, 'Application of the Genetic Algorithm to Automatic Program Generation'. Master's thesis, University of Idaho, Moscow, ID.
18. Holland, J.: 1975, *Adaption in Natural and Artificial Systems*. The University of Michigan Press.
19. Jones, S. and C. Hinde: 2007, 'Preservation of schemata using introns'. In: G.M.Coghill (ed.): *Proceedings of the 2007 workshop on Computational Intelligence*. University of Aberdeen, London, England.
20. Kantschik, W. and W. Banzhaf: 2001, 'Linear-Tree GP and its comparison with other GP structures'. In: J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (eds.): *Genetic Programming, Proceedings of EuroGP'2001*, Vol. 2038 of *LNCS*. Springer-Verlag, Berlin, pp. 302–312.

21. Kantschik, W. and W. Banzhaf: 2002, 'Linear-Graph GP - A new GP Structure'. In: J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (eds.): *Genetic Programming, Proceedings of EuroGP'2002*, Vol. 2278 of *LNCS*. Springer-Verlag, Berlin, pp. 83–92.
22. Keijzer, M. and M. Cattolico: 2002, 'An example of the use of context-sensitive constraints in the ALP system'. In: *Grammatical Evolution Workshop, GECCO 2002*. Morgan Kaufmann, San Francisco, CA 94104, USA.
23. Keller, R. E. and W. Banzhaf: 1996, 'Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes'. In: *Genetic Programming*. The MIT Press, Cambridge, MA:.
24. Koza, J.: 1989, 'Hierarchical Genetic Algorithms Operating on Populations of Computer Programs'. In: N. Srindharan (ed.): *Proc. of the 11th Intern. Joint Conf. on Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA 94104, USA, pp. 768–774.
25. Koza, J.: 1992, *Genetic Programming: On the Programming of Computers by means of Natural Selection*. MIT Press.
26. Koza, J.: 1994, *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.
27. Koza, J., D. Andre, F. Bennett, and M. Keane: 1999, *Genetic Programming 3: Darwinian Invention and Problem Solving*. San Francisco, CA 94104, USA: Morgan Kaufmann.
28. Montana, D.: 1995, 'Strongly Typed Genetic Programming'. *Evolutionary Computation* **3**(2), 199–230.
29. Niehaus, J. and W. Banzhaf: 2001, 'Adaption of Operator Probabilities in Genetic Programming'. In: J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (eds.): *Genetic Programming, Proceedings of EuroGP'2001*, Vol. 2038 of *LNCS*. Springer-Verlag, Berlin, pp. 325–336.
30. Nordin, J. P.: 1995, 'A Compiling Genetic Programming System that Directly Manipulates the Machine code'. In: J. K. Kinnear (ed.): *Proceedings of the Sixth International Conference of Genetic Algorithms*. MIT Press, Cambridge.
31. O'Neill, M., T. Brabazon, C. Ryan, and J. J. Collins: 2001, 'Developing a Market Timing System using Grammatical Evolution'. In: H. Beyer, E. Cantu-Paz, D. Goldberg, S. Parmee, and D. Whitley (eds.): *Proceedings of GECCO 2001*. Morgan Kaufmann, San Francisco, CA 94104, USA.
32. O'Neill, M. and C. Ryan: 1999, 'Evolving Multi-line Compilable C Programs'. In: R. Poli, P. Nordin, W. B. Langdon, and FogartyT.C. (eds.): *Genetic Programming, Proceedings of EuroGP'99*, Vol. 1598 of *LNCS*. Springer-Verlag, Berlin, pp. 83–92.
33. Paterson, N. and M. Livesey: 1996, 'Distinguishing Genotype and Phenotype in Genetic Programming'. In: J. Koza, D. Goldberg, D. Fogel, and R. Riolo (eds.): *Late-breaking Papers, Genetic Programming 1996: Proceedings of the First Annual Conference*. MIT Press, Cambridge, MA:.
34. Paterson, N. and M. Livesey: 1997, 'Evolving caching algorithms in C by genetic programming'. In: *Genetic Programming 1997*. Morgan Kaufmann, San Francisco, CA 94104, USA, pp. 262–267.
35. Perkis, T.: 1994, 'Stack-based Genetic Programming'. In: *IEEE World Congress on Computational Intelligence*. IEEE Press, pp. 148–153.
36. Ross, B.: 1999, 'Logic-based Genetic Programming with Definite Clause Translation Grammars'. Technical Report CS-99-92, Brock University.

37. Ryan, C., J. Collins, and M. O'Neill: 1998a, 'Grammatical Evolution: Evolving Programs for an Arbitrary Language'. In: W. Banzhaf, R. Poli, M. Schoenauer, and T. Fogarty (eds.): *EuroGP '98*. Springer, Berlin, pp. 83–95.
38. Ryan, C., J. Collins, and M. O'Neill: 1998b, 'Grammatical Evolution: Solving Trigonometric Identities'. In: *Proceedings of Mendel '98: 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks and Rough Sets*. Technical University of Brno, Brno, Czech Republic, pp. 111–119.
39. Smith, S.: 1983, 'Flexible Learning of Problem Solving Heuristics Through Adaptive Search'. In: A. Bundy (ed.): *IJCAI*. William Kaufmann, Los Altos, CA:, pp. 422–425.
40. Syswerda, G.: 1989, 'Uniform Crossover in Genetic Algorithms'. In: J. Schaffer (ed.): *Proceedings of Third International Conference on Genetic Algorithms*. Morgan Kaufmann, San Francisco, CA, USA, pp. 2–9.
41. Teller, A. and M. Veloso: 1995, 'PADO: Learning tree structured algorithms for orchestration into an object recognition system'. Technical Report CMU–CS–95, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
42. Wall, L., T. Christiansen, and R. Schwartz: 1996, *Programming Perl*. O'Reilly & Associates Inc., second edition.
43. Whigham, P.: 1995, 'Grammatically-based Genetic Programming'. In: J. Rosca (ed.): *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*. National Resource Laboratory for the Study of Brain and Behavior, pp. 33–41.
44. Wineberg, M. and F. Oppacher: 1994, 'A representation scheme to perform program induction in a canonical genetic algorithm'. In: Y. Davidor, H.-P. Schwefel, and M. Schwefel (eds.): *Parallel problem solving from nature III*, Vol. 866 of *Lecture Notes in Computer Science*. Springer, Berlin, pp. 86–96.
45. Wineberg, M. and F. Oppacher: 1996, 'The benefits of computing with Introns'. In: J. Koza, D. Goldberg, D. Fogel, and R. Riolo (eds.): *Genetic programming 1996: Proceedings of the First Annual Conference*. The MIT Press, Cambridge, MA:, pp. 410–415.
46. Withall, M., C. Hinde, and R. Stone: 2004, 'Evolving the user interface'. In: M. Withall and C. Hinde (eds.): *Proceedings of the 2004 UK Workshop on Computational Intelligence*. Loughborough University, Loughborough, pp. 86–96.
47. Yu, T.: 2001, 'Polymorphism and Genetic Programming'. In: J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon (eds.): *Genetic Programming, Proceedings of EuroGP'2001*, Vol. 2038. Springer-Verlag, Berlin, pp. 218–233.