Evolving Programs with Parameters and Loops

Gayan Wijesinghe and Vic Ciesielski

***Abstract*— At the current state of the art, genetic programs do not contain two constructs that commonly occur in programs written by humans, that is, loops and functions with parameters. In this paper we describe an investigation into the evolution of programs for a problem that can only be solved by evolving a parameterised program with one or more loops. We provide training examples of the desired program behaviour for a number of problem sizes and require the evolution of a program *P(n)* that will give the correct output for any value of *n*. We have chosen a problem, that of reproducing a binary string to a given number of bits, that can be made harder or easier by adjusting various aspects of the formulation. We are interested seeing in which formulations lead to success and which do not. We conclude that programs with parameters and loops can be successfully evolved if the search space is appropriately restricted by (1) grammars which restrict the possible program structures, (2) limits on program depth and (3) limits on the range of random constants.**

I. Introduction

While there has been considerable success in the use of genetic programming (GP) techniques on a large number of problems, the evolved programs lack some of the important characteristics normally found in the kinds of programs written by human programmers. In particular, parameterised programs with evolved loops have been extremely limited.

There are many factors that make evolving programs with loops practically prohibitive. For example, crossover and mutation operations too often destroy valid loop structures than help evolve them, unless constraints are used to force the evolution of only valid loops. Even with such restrains, a valid loop must simultaneously ensure the initialisation of loop variables, failing of the loop condition after the required number of iterations and the synchronisation of functions in the loop body with the loop variables. The complexity introduced to the search by the last requirement alone is so high that, to obtain a correct program with loops for even a small, apparently simple problem requires massive increases in computational time. A single program with an infinite loop, can upset the fitness evaluation process of the entire population per each generation. In reality, the frequency of infinite loops in populations are very high. For these reasons, among others, GP researchers and practitioner tend to avoid evolving programs with loops.

There are two main ways of dealing with situations where repetitive behaviour is required. The first is to put the loop control in the environment and only evolve the body. For example, in a typical solution to the Santa Fe Ant problem [1], the evolved program is called over and over again until the problem is solved or the maximum number of steps is exceeded. The second is to put the repetitive behaviour into a terminal. For example, in [2] the loops needed to calculate image features that are embedded in a number of terminals.

Gayan Wijesinghe and Vic Ciesielski are with The School of Computer Science and Information Technology, RMIT University, GPO Box 2476V, Melbourne Victoria 3001 (email: gayan.wijesinghe,vic.ciesielski@rmit.edu.au).

Solution generality is also a characteristic that is highly sought after. For the purposes of this paper we define generalisation as the ability of an evolved program to solve various size instances of a problem, when the problem size is specified to the program by a parameter n. The issues encountered when evolving generalised programs rise in the learning process, where generalisation has to be promoted with a limited number of examples, and in the testing process, where generalisation should ideally be tested on all sizes of the problem. A common approach in the field is evolving a program for each size of a problem and then externally selecting the program to run depending on the size. As it is infeasible to evolve a program for each size of a problem, the use of GP has been limited in such problems.

TABLE I
Train and Test Examples of the Desired Program Behaviour

Training $P(n)$	
n	Given Example
3	101
4	1011
5	10110
6	101101
...	...
15	101101101101101

Testing $P(n)$	
Program call	Output
P(16)	1011011011011011
P(17)	10110110110110110
...	...
P(63)	10110110110110110...

In this work, we investigate how the above issues relating using loops, parameters and generalisation can be addressed. We test our approaches by evolving programs that can correctly learn and output repeating binary string patterns to any size. For example, consider the binary pattern “101101101”, which has a repeating unit of length 3 (i.e. $U = 3$), “101”. As shown in Table I, we wish to evolve a parameterised program, $P(n)$, from a finite number of examples of the pattern at various lengths and then expect this program to be correct for any n.

We are interested in knowing under what conditions we can have success when using loops and parameters in GP. We chose this problem because we can manipulate its difficulty.

A. Research Questions

The overall goal of our work is to investigate the use of loops and parameters in GP and whether they can be used for generalisation. In particular, we are interested in answering the following research questions:

1) How can loops and parameters be incorporated into tree-based genetic programming?
2) How can the learning task be presented at different levels of difficulty?
3) How can fitness be used to promote generalisation and how confident can we be that a program will give correct output for any n?

To answer the above questions, we break our pattern learning problem down to two sub-problems. In the first sub-problem, we investigate whether correct programs can be evolved for whole repetitions of the repeating unit. We refer to these as "whole repetitions" experiments. In the second sub-problem, we investigate whether programs can be evolved to output the correct patterns at any length. These experiments, we refer to as "repetitions with remainder". Even when limited to learning only whole repetitions, we expect the correct learning of the repeating unit and the emergence of a general solution to be very difficult. For this reason, we use three problem formulations at different levels of difficulty. We call these, "easy", "medium", and "hard" formulations. The levels of difficulty are controlled by imposing various restrictions on grammars that specify the kinds of programs that can be evolved.

It is important to note that we require the evolved program to perform the entire task correctly, without any help from the environment. The program must output exactly the right number of bits. It is not acceptable, for example, for the program to output a long string which is trimmed to the right number of bits by the environment.

II. Related Work

In this section, we discuss prior work that has been done in relation to representing loops, parameters and generalisation in genetic programming.

One of the earliest works relating to parameterisation was published by Koza, using a method called automatically defined functions (ADF) [3]. In this method, program trees employ a strict tree structure definition to evolve separate sub-trees as reusable code and one sub-tree to evolve the function-calls that ultimately produce the output. Koza then extended this work to automatically defined loops (ADL) [4]. The concept behind the ADL representation within GP trees is similar to that of the ADF representation. The loop component is evolved as a separate sub-tree and is then used by another subtree which produces the final output.

None of the above work concentrated on generalisation, however, in his 4th book on GP, Koza describes a method to evolve parameterised topologies for electronic controllers and circuits [5]. This work does claim generality of the found solutions, however, the outputs produced by their formulations are for a fixed problem size. In contrast, in our work, the problem size is specified to the program by the parameter n.

Langdon used a type of 'forwhile' loops in ADFs to evolve the operations of a list data structure [8]. In his work, the fitness is based only on the tests that have been passed by a program, ensuring that a program need to completely and accurately learn every case to be a solution. In order to achieve success, they used a strong restriction on the maximum number of iterations of 32. In comparison, we use a less restrained structure to our programs, to see the possibility of evolving a general program under less restrained conditions.

Kinnear showed a way to use loops to evolve a sorting algorithm [6]. The author used a construct similar to a FOR-loop with an index and managed to produce solutions for sorting arrays of lengths up to 30. Here, many training cases were used to improve solution generality but there was no mechanism in the in the fitness evaluation or technique presented on how training examples should be used. Therefore, generality was not guaranteed. Later Ciesielski and Li used a non-indexed FOR-loop construct in GP to evolve a sorting algorithm for a fixed-length array [7]. They did not focus on generality but rather the correctness by training solutions on every permutation of the input data at a fixed size. In our work, we follow a similar strategy by using a large number of examples but they are of varying sizes and the majority of them are in the testing process, to increase our confidence in solution generality.

Li and Ciesielski used a domain dependant form of FOR-loops to solve a two class small binary image classification problem [10]. In their method, the loop traverses coordinates within images to perform feature calculations which are then used for classification purposes. This work has been extended by Wijesinghe and Ciesielski to classify relatively large grey-level images that were transposed and rotated [2]. In both methods, the classifiers that used loops were more successful than the solutions that did not use loops. However, none of these works focused on solving the generalisation problem that we explained in the introduction.

A popular category of problems that require iteration is maze traversal, such as the Santa Fe Ant problem. Several authors have published work on this using loops but none of these works concentrated on using parameters or obtaining general solutions [7], [11]. For example, representing a maze traversal the way we have represented our binary pattern learning problem would require a program to evolve, that would not perform any illegal operations (e.g. walking into walls, etc.), and still be general for any scale of the maze.

III. Using Loops and Parameters in Genetic Programming

In this section we first discuss the general concept of representing loops in GP. We then describe the specific representations that we use for the programs in different levels of difficulty, applying to the whole repetition experiments. The modifications made to these representations for the remainder experiments are then discussed before

finally, discussing the approach to fitness determination for promoting generalisation.

A. Representing Loops and Parameters

In order to answer our first research question, on representing loops in GP, we look at how loops are used in programming languages. A looping construct in a procedural programming language such as C contains several components. There are many ways to obtain the same effect but consider the following C language code example:

```
for (index=0; index<times; index++)
   printf("%d ", index);
```

The above `for` loop contains components for the initialisation and modification (e.g. incrementing) of the `index` variable, a continuation condition and a loop body block of code that is executed at each iteration. If we assume that the above is the underlying logic of a typical loop, which is the `index` starts at 0 and is only incremented by 1, the above continuation condition means that the loop will iterate `times` number of times, for any positive value of `times`. For the loop to function correctly, all of these components must (1) be present and (2) have the values properly initialised and updated. In tree-based GP the probability of having all of these components present in an evolved individual is very small, unless specialised crossover and mutation operators are used. For this reason we use a simplified loop construct in which the index initialisation, modification and the left hand side of the continuation condition are kept constant and we can simply evolve `times`, as a function of the problem size n to determine the number of times the loop will iterate. The loop construct will then ensure that the `index` is initialised to 0 and is incremented by 1. Additionally, with a rich set of functions, the instructions in the body can then be evolved to use the `index` correctly for the range of values it will hold. We express our basic GP loop formulation as follows:

```
(LOOP TIMES BODY)
```

The `LOOP` is the parent node and `TIMES` and `BODY` are its subtrees. The `TIMES` subtree is a function of the parameter n, and can consist of basic mathematical operators such as addition, subtraction, division and multiplication as well as random numbers. It will then evaluate to a single number, with respect to the n parameter, which is the number of times the loop will iterate. The `BODY` subtree will contain one or more functions that will operate (produce side-effects) on a data structure (i.e. the output string) with respect to the `index`, which is maintained by the `LOOP`. By using strongly typed GP, we can ensure that crossover and mutation will not, say, crossover a `TIMES` subtree with a `BODY` subtree.

The specifics of how we formulate our programs, following the said approach, are discussed below.

B. Levels of Difficulty and Reproducing Whole Repetitions

In this section we describe our approach for answering research question 2, on how we can represent the learning task at different levels of difficulty, in order to understand how generalisation occurs. We have identified several ways in which we can manipulate our problem and system in order to vary the difficulty of the search.

We first consider the learning task itself, which is evolving a program that uses a parameter n and looping constructs to correctly reproduce a learnt pattern exactly to a given length, specified by the parameter. From a learning perspective, it is difficult to train such a program to the accuracy of a single bit. For this reason, we broke our learning task down to two sub-problems.

The number and types of functions and terminals used in the programs have a direct connection to search difficulty, as they increase the size of the search space. We use strongly-typed GP and grammars to structure our GP programs and through these grammars, we are also able to control the complexity of solutions by allowing and disallowing certain mathematical operations. Another such aspect that we can control in the programs is the range of random numbers that are generated for terminals. By manipulating these attributes to vary the search difficulty, we present 3 formulations for each sub-problems named "easy", "medium" and "hard", which we describe in detail further below.

The setup of the GP environment too can affect the search difficulty. Particularly, increasing the maximum depth limit leads to a much larger number of tree shapes, directly affecting the search space. Therefore, this too is an aspect that we vary for each of our experiments (refer table II).

1) "Easy" Difficulty Formulation:

The intuition behind all our program formulations is to include a construct that can output a learned substring (i.e. the repeating unit) in sequence and another construct that can repeat the said construct a number of times to generate the final pattern. In particular, we use a function called `PATCH` that will learn and output an entire substring of the pattern. For this, the boundaries of the repeating unit within the largest example in the training set are learnt in the sense of "extracting" it from the examples at hand. This then will be repeated by a `LOOP`, which is a function of the parameter n. Below is a C language program that outputs a substring from the longest example in the `examples` array:

```
for (i=0; i<n/3; i++)
   for (p=0; p<3; p++)
      printf("%c",examples[longest][p]);
```

In our GP representation, we wish to obtain a tree-based program that is equivalent to the above such as:

```
LOOP (/ N 3) (PATCH 3)
```

In this "easy" formulation, we wish to reduce the size of the search to a bare minimum, while evolving only the vital parts of the program. We expect that the search will be so constrained that virtually every run will be successful and this will validate our approach. We assume that the repeating unit of the pattern in the example starts at position 0, minimising the need to search for the starting position. The length of the substring (which is the "3" under `PATCH`, in the above example) is then searched for and is counted from position 0.

```
LOOP         -> "LOOP" TIMES BODY
TIMES        -> TIMES / TIMES | RAND_INT | N
BODY         -> PATCH
PATCH        -> "PATCH" PATCH_LENGTH
PATCH_LENGTH =  randNum %
                (maxExampleLength/(2*unitLength)))
                * unitLength
RAND_INT     =  a random integer
```

Fig. 1. Grammar for "easy" difficulty formulation.

The PATCH function performs the functionality of the inner for loop shown in the C language version, reducing the complexity of the search.

The grammar in Figure 1 describes this formulation in more detail. The value of RAND_INT is in the range of 0 and the size of the largest example in the training set. However, we use a subset of this range for PATCH_LENGTH. In particular, we restrict the length of the patch to numbers that are multiples of the repeating unit length (using prior knowledge of the pattern) within the first half of the largest training example. The intuition behind this is in twofold. First, we know that the largest training example should have at least two repetitions of the repeating unit of the pattern or even a human will not be able to learn the entire pattern. So the length of the extracted patch or substring will be at most half this size. Then, the reason that we have this length of the patch as multiples of the repeating unit is in order to reduce the search space, just in this simplified formulation. We also limit the maximum tree depth to 4, which is the minimum necessary to evolve a correct program in this formulation.

To re-iterate, in this formulation, we wish to make the search space as small as possible, in order to focus on achieving generalised solutions. These restrictions are relaxed in the "medium" and "hard" formulations.

2) "Medium" Difficulty Formulation:
In "Medium", we remove some of the assistance we provided to the search in the "easy" formulation and see whether generalisation can still take place.

```
LOOP         -> "LOOP" TIMES BODY
TIMES        -> TIMES / TIMES | TIMES * TIMES |
                TIMES + TIMES | TIMES - TIMES |
                RAND_INT | N
BODY         -> PATCH
PATCH        -> "PATCH" PATCH_START PATCH_LENGTH
PATCH_START  =  randNum % (maxExampleLength/2)
PATCH_LENGTH =  randNum % (maxExampleLength/2)
RAND_INT     =  a random integer
```

Fig. 2. Grammar for "medium" difficulty formulation.

As shown by the grammar in Figure 2, one of the key changes in this formulation is the TIMES subtree, where integer multiplication, addition and subtraction operations are now allowed. We also increase the difficulty of the problem by searching for the starting location of the patch, PATCH_START. Now, both PATCH_START and PATCH_LENGTH are searched for as random constants in the range of 0 and half the length of the longest example, following our rationale described in the previous section. Furthermore, we increase the maximum depth limit from 4, in the "easy" formulation, to 7.

3) "Hard" Difficulty Formulation:
The formulations what we have described so far have only allowed a program to contain a single looping construct. In this formulation, we extend the "medium" formulation by allowing multiple LOOP functions, as described by the grammar in Figure 3.

```
PROG -> PROG PROG | LOOP
LOOP -> "LOOP" TIMES BODY

[Rest of the grammar is the same as in figure 2]
```

Fig. 3. Grammar for "hard" difficulty formulation.

Here, a number of LOOP subtrees can be executed in series. We expect programs of this formulation to contain loops that output different parts of the pattern string, in sequence. A possible program for a pattern with a repeating unit length of 2 in this formulation is:

```
PROG
 (LOOP (/ N 4) (PATCH 0 2))
 (LOOP (/ N 4) (PATCH 0 2))
```

In the above, the first loop output the first half of the pattern and the second outputs the last. Of the three formulations described, this is expected to be the most difficult.

C. Levels of Difficulty and Repetitions with Remainder

In the previously discussed formulations of the problem, the input to the program, n, was required to be a multiple of the repeating unit string. In this section, we wish to extend our programs so that they will work for arbitrary values of n. In particular, if a certain n value requires less than one whole repetition of the unit string, the programs must be able to produce any whole repetitions and the remainder of the output to match the length exactly to n. It is equally valid to do the whole repetitions first and the remainder last as it is to do the remainder first and the whole repetitions last.

We implement the remainder producing mechanism as an extension to the grammars that describe our previous formulations. For example, the following grammar describes the "easy" formulation with the new mechanism:

```
PROGRAM        -> LOOP REMAINDER
REMAINDER      -> "REMAINDER" REMAINDER_FUNC
REMAINDER_FUNC -> REMAINDER_FUNC % REMAINDER_FUNC |
                  RAND_INT | N

[Rest of the grammar is the same as in figure 1]
```

Fig. 4. Remainder handling mechanism for "easy" formulation.

In the above, REMAINDER_FUNC contains the arithmetic that determines the length of the remainder to output. Once the LOOP has completed its output, REMAINDER_FUNC number of characters from the largest example (similar to how PATCH works) is output, starting from position 0 in the largest example. We made this same assumption regarding the patch starting from position 0 in the whole repetition experiments for the "easy" difficulty formulation as well.

Thus, there is no additional domain knowledge introduced for the remainder experiment with the "easy" difficulty formulation.

Following our strategy of increasing the complexity of the arithmetic possible with the functions and terminals allowed in the "medium" and "hard" formulations in the whole repetitions experiments, we also increase the complexity of the REMAINDER mechanism for the "medium" and "hard" formulations in our remainder experiments as follows:

```
PROGRAM         -> LOOP REMAINDER | REMAINDER LOOP
REMAINDER       -> "REMAINDER"
                   REMAINDER_START REMAINDER_FUNC
REMAINDER_FUNC -> REMAINDER_FUNC % REMAINDER_FUNC |
                   REMAINDER_FUNC * REMAINDER_FUNC |
                   REMAINDER_FUNC / REMAINDER_FUNC |
                   REMAINDER_FUNC + REMAINDER_FUNC |
                   REMAINDER_FUNC - REMAINDER_FUNC |
                   RAND_INT | N
REMAINDER_START-> RAND_INT

[Rest of the grammar is the same as in figures 2 and 3]
```

Fig. 5. Remainder handling mechanism for "medium" and "hard" formulations.

In the above, the REMAINDER_START is used to find the starting location of the remainder since, in the "medium" and "hard" formulations, the patch starting location is evolved (in contrast to being fixed at 0, in the "easy" formulations).

Additionally, for the "hard" formulation, we allow multiple LOOP and REMAINDER pairs in the programs, increasing the complexity, as follows:

```
PROG2   -> PROG2 PROG2 | PROGRAM
PROGRAM -> LOOP REMAINDER | REMAINDER LOOP

[Rest of the grammar is the same as in figure 5]
```

Fig. 6. Multiple loop and remainder pairs allowed in the "hard" formulation.

This is similar to allowing multiple loops in the "hard" formulation in the whole repetition experiments.

D. Fitness Measurement for Generalisation

To reiterate, our problem requires a parameterised program $P(n)$ to be evolved and be able to reproduce a repeating binary string to a user specified length n. Any variations from the exact target would make the program invalid. To make the evolved solutions conform to these strict requirements and generalise, we employ a fitness measurement strategy that focuses on examples and their presentation. To generate the training examples, we take the longest example that we have and then extract substrings from it at various lengths, starting at the first position.

Next, we focus on the presentation of data in the training phase. It is usual to employ a strategy such as evaluating the success on a large number of training examples and then averaging the result as the fitness of an individual. However, generalisation, in the context of this paper, means obtaining success on every training example, from the smallest to the largest, as well longer testing examples. Our preliminary experiments have shown that averaging success rates as fitness does not encourage systematic learning, so we first present the individual with the shortest example and only if this is successful is the next example presented. This strategy is repeated until the program is able to solve all the training examples. The fitness of an individual is the number of examples in the training set that it was not able to solve in this manner, and we want the GP search to minimise this number.

Our rationale here is that, programs that are not able to reproduce the pattern from the smallest to largest lengths, in sequence, are likely to get stuck at a sub-optimum in the search space and not improve. In contrast to averaging the success, this strategy also prevents a program from "unlearning" cases that it has learnt to solve previously as the search progresses.

In the whole repetition formulation, where the learning is restricted to learning and reproducing whole repetitions of a pattern, if we say the unit length of the pattern is U, we train the population on every multiple of U in the first $U*(U+1)+U$ bits of the pattern, starting from U. From our preliminary tests, we have seen that this number of examples is adequate for obtain correct programs. For example, for the pattern "101...", where $U=3$, we would use only the cases where the n value is a multiple of 3. As $U*(U+1)+U=15$ is the length of our longest training example, it is also the upper bound of the random number range in the program grammars.

Once a program is fully trained, only then we proceed to the testing phase. In testing, we are interested in seeing the extent of the generalisation obtained during the training phase by testing the program on outputs that are longer than the presented training examples. A fully trained program is checked to see if it can reproduce the learned pattern to strings that are up to and including 5 times longer than the largest training example used, i.e. outputs that are up to $5*U*(U+1)+U$ long.

For example, in the whole repetition experiments for a pattern with $U=25$, the population is trained on example lengths that are all the multiples of 25 up to 675 bits long. Once a program is fully trained, it is then tested on all the multiples of 25 that are up to 3275 bits long (131 examples). Only if the program manages to correctly and sequentially pass all the training and testing cases do we refer to it as a successful run.

In the remainder formulations, the programs are trained on every length, incremented by 1 bit at a time, starting from 1 whole repetition. Table I shows these training cases for the pattern "101". The remainder approach uses a significantly larger number of examples. For example, for a pattern with $U=25$, programs will be tested on 3250 examples, whereas the whole repetition approach will only use 131.

IV. Analysis of Results and Generalisation

The differences in representational complexity between the two sub-problems required us to configure the GP system differently. For the whole repetitions the search space is smaller. For this reason, we set our maximum generation

limit to 16,000 generations (or 1,600,000 evaluations). The remainder handling approach, in the remainder experiments, introduces a much larger search space due to the additional mechanisms introduced in the grammars, so we used a maximum generation limit of 64,000 generations (or 6,400,000 evaluations). Training runs were carried out for each of the 6 formulations using patterns with U values of 2, 10, 25 and 50. For each such U value (except $U = 2$), 3 different strings were chosen from a pool of 50,000 randomly generated binary unit strings for each size. The 50,000 random strings were sorted on standard deviation and the 3 were chosen to represent patterns with low, medium and high values. Therefore, a total of 60 experiments were performed.

TABLE II

GP SETUP FOR EXPERIMENTS

Parameter	Whole Repetition	With Remainder
Runs	40	60
Population size	100	100
Mutation rate	0.23	0.23
Crossover rate	0.75	0.23
Elitism rate	0.02	0.02
Max. generations	16000	64000
Max. depth		
Easy	4	5
Medium	7	8
Hard	8	9

The final success rates for each sub-problem are shown in Table III. Considering first the performance of the whole repetition experiments, the success rates reduced as the problem difficulty increased from "easy" to "hard", as expected. Also, a similar pattern was seen for every level of difficulty, when strings of different lengths were used – increasing the repeating unit length (U) meant that fewer runs were successful. This is also expected as the probability of finding the right patch length decreases with increasing unit length.

The results suggest that it is possible to evolve generalised, parameterised programs containing loops, but only if steps are taken to restrict the search space.

It was interesting to note that in the "hard" formulation experiments, where multiple loop structures were allowed, while early generations contained programs with several loops, all of the successful runs generated programs containing only one loop.

Analysis of the successful evolved programs for the whole repetitions formulations indicates that the programs will give correct behaviour for any value of the parameter n. The following is a typical successful program for $U = 2$ from the "easy" runs:

```
(LOOP
   (/ N 2)
   (PATCH 2))
```

It is clear from the text of the program that it will give the correct output for any value of the parameter n. The following program was evolved for a string size 50 in the "hard" formulation:

```
(LOOP
   (/ N (/ 2185 43))
   (PATCH 1050 50))
```

In the above, simplification of the TIMES subtree gives (/ N 50) which is the desired function and the patch length too is correct at 50. The search has chosen to extract the patch from the largest example from location 1050, which is correct since it is a multiple of 50, starting from 0. Therefore this is a correct program that can reproduce the learned pattern to any n, as long as its value is a multiple of the repeating unit.

The following is a much more complex program evolved for the whole repetition experiments, for a string size of 10 in the "hard" formulation:

```
(LOOP
   (+ (/ N (/ 95 9)) (* (- (+ (/ N 9) 477) 9)
    (- (- N (/ (+ 973 19) (/ 692 830))) N)))
   (PATCH 120 10))
```

After simplification, the program becomes:

```
(LOOP
   (/ N 10)
   (PATCH 120 10))
```

Again it is clear that this program is correct for any n.

The runs for the remainder formulations were considerably more interesting. In the "easy" formulation most of the runs were successful and generated correct programs. The following is a typical program for $U = 2$:

```
(LOOP
   (/ N 2)
   (PATCH 2))
(REMAINDER
   (% N 2))
```

However, there were very few successful runs with the "medium" and "hard" formulations. Most of the runs were trapped in local minimum where program behaviour was correct for values of n in the training data but not correct for larger values. The grammars that we used we expected to evolve programs in which the whole repetitions would be produced by the main loop and the remainder to be produced by the REMAINDER mechanism. However, in almost all of the evolved individuals, the loop was not repeating all the whole repetitions of the repeating unit and the remainder mechanism completed the output for it. Below is a sample program from the "medium" difficulty level for the pattern "10..." ($U = 2$), demonstrating this behaviour:

```
(LOOP
   1
   (PATCH 0 1))
(REMAINDER
   1
   (- N 1))
```

In this program, for any value of "N", the loop iterates only once ((LOOP 1...) and writes out the first bit starting from position 0 ((PATCH 0 1)). The remainder length calculation function has evolved to write out the remaining $n - 1$ characters, starting from 1, which is the next position ((REMAINDER 1 (- N 1))). Not all programs were as simple as the above but through manual verification we found

the logic of the behaviour to be the same. This behaviour was not observed in the "easy" experiments. The evolved programs here all produced the output string using the loop and the remainder mechanisms as intended.

Through analysis of the evolved programs we found that the reason for the unexpected behaviour was the more complex arithmetic permitted in the TIMES subtree and the increased depth limit in the "medium" and "hard" formulations. By changing the fitness function to enforce an upper limit of U on the value returned by the REMAINDER_FUNC subtree we were able to get a larger number of successful runs. If the REMAINDER_FUNC subtree did not evaluate to a number less than or equal to the unit length U, the program was considered as having failed fitness evaluation. These results are shown in Table III. The overall trends are similar to the whole repetitions results. Below is a valid program found after this modification, in a "medium" difficulty experiment for $U = 10$ pattern:

```
(LOOP
   (/ N 10)
   (PATCH 30 10))
(REMAINDER
   60
   (% N 10))
```

Note that the starting positions for the patch (30) and the remainder (60) both are valid since they are within the bounds of the largest example. The following is a much more complex program that was found in the same experiment:

```
(LOOP
   (/ N 10)
   (PATCH 210 10))
(REMAINDER
   240
   (% N (% (/ (+ (- 314 77) 553) (+ (- 394
    368) 53)) (+ (+ 276 107) 328))))
```

And after simplification of the above we can obtain:

```
(LOOP
   (/ N 10)
   (PATCH 210 10))
(REMAINDER
   240
   (% N 10))
```

There are a number of other methods that use domain knowledge that we could have used to deal with the local optimum problem, for example, limiting the remainder to the structure defined in the "easy" formulation, limiting the tree depth and forcing the loop to produce more output than the remainder mechanism for values of n larger than the repeating unit. However, all of these methods involve the use of domain specific knowledge. There does not appear to be a general method of dealing with a situation like this. Limiting the number of iterations of a loop by a global "maximum permitted iterations" parameter, for example, would not help.

A. Confidence in Generalisation

We have high confidence that the programs evolved in successful runs will be general, that is, give the correct output for any value of the parameter, n. This confidence comes

TABLE III
COMPARISON OF SUCCESS RATES

Formulation	Unit String Length	Unit String Count	Success Rates: Whole repetitions	Success Rates: Repetitions with remainder
Easy	50	3	40/40	58/60
	25	3	40/40	60/60
	10	3	40/40	59/60
	2	1	40/40	60/60
Medium	50	3	10/40	0/60
	25	3	17/40	1/60
	10	3	35/40	7/60
	2	1	40/40	15/60
Hard	50	3	3/40	0/60
	25	3	14/40	0/60
	10	3	32/40	6/60
	2	1	40/40	6/60

from (1) the training strategy, (2) testing the programs on many values of n larger than any used in training, and (3) manual simplification and verification.

During the training, we forced the individuals in the population to solve problems from the smallest size to a large size without getting any wrong. In the runs, we observed that programs were fairly slow to solve the first training example but after they progressed through the first few examples, they were very quick to solve the rest of the examples. This can be seen as a steep improvement in fitness as shown by the graph in figure 7. The Y-axis of this figure shows the number of training cases that an individual is getting wrong. This figure suggests that once a good program is found for the number of examples processed so far, it is also good for examples of a larger size, an furthermore, once an example is learned, it is not "unlearned" in a later generation.

To evaluate the generality of individuals that achieved perfect performance on the training data we tested them on example sets that were up to 5 times larger than the training sets. An individual needed to get these additional cases correct before it was accepted as a solution. While no practical amount of brute force testing guarantees the correctness of logic, the fact that individuals exhibit the

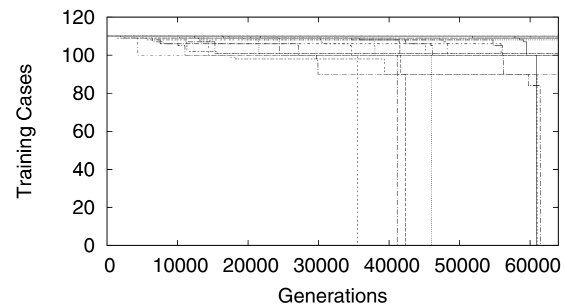

Fig. 7. Steep improvement curve shown by programs that generalise.

correct behaviour on many examples larger than the training ones adds to our confidence that they are correct.

We inspected the program trees of many solutions and manually verified that they were correct. For the simpler solutions this was quite straight forward. For the more complex solutions manual simplification of some of the arithmetic expressions was required. In all the cases we examined the logic was correct. The simpler solutions also conform to Occam's razor, which suggests that simpler solutions are better than more complex ones and are likely to be correct.

While we have no formal proof that the evolved solutions are correct, from the kinds of analyses described in this section we have high confidence that they are.

V. Conclusions

Our main goal in this work was to determine whether we could evolve parameterised programs containing loops that would be correct for any value of the parameter n, not just the training examples. We have shown that this can be successfully done if the evolutionary search is constrained by limits on the range of random constants, limits on program depth, and grammars which restrict the possible program structures.

To answer the first research question, we have incorporated loops and parameters into tree-based genetic programming by taking traditional procedural programming loops as a metaphor and simplifying them to avoid syntactical complications that can arise when used in GP evolution. We have shown that a looping construct can be represented in a tree form with two subtrees. The first sub-tree is the number of times the loop will iterate and the second subtree is the body of the loop containing the actions that will be repeated.

In order to investigate the conditions under which generalisation would be successful, it was necessary to formulate the learning task at different levels of difficulty. This was our second research question and we achieved this through several means. First we divided the binary pattern string learning task into two sub-problems – (1) the parameter n was a multiple of the repeating unit and (2) the parameter could be any positive integer. As expected the success rate was higher in situation (1). For each of these sub-problems we formulated learning tasks of "easy", "medium" and "hard" difficulty by adjusting the program grammars, random number ranges, tree depths and repeating unit lengths. As expected, the more difficult formulations had lower success rates. Overall we obtained at least one solution for 21 of the 24 types of experiments that we performed.

Our third research question was on how fitness measurement could be used to promote generalisation and how confident we can be about the generality of the found programs. We incorporated two strategies to address fitness measurement question: (1) providing multiple examples of the same pattern at different lengths and (2) forcing the programs to learn one example at a time from the training set, starting from the one of smallest length. The first strategy ensures that there are enough examples; the second ensures that once a program learns to solve a particular example, it will not lose this ability in a later generation in the evolution. Our experimental results show that, while success rates are can become low for the more difficult formulations of the learning tasks, programs that were able to sequentially solve the first few examples were able to very quickly learn the rest and fully generalise in most runs. This observation raises our confidence in this approach. Our confidence was further increased by the fact that all of solutions that we analysed manually were correct.

A. Future Work

In this work, the evolved programs were dominated by the looping construct. In fact, in most of the formulations the loop was forced to be at the root. We would like to look at problems where the loop is a small part of the program to be evolved. Another avenue is extending from learning patterns in 1 dimension to patterns in 2 dimensions. We would also like to see how successful our methodology will be in similar learning problems requiring parameters and loops.

References

[1] W. B. Langdon and R. Poli, "Why ants are hard," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, K. et al., Ed. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, 22-25 1998, pp. 193–201.

[2] G. Wijesinghe and V. Ciesielski, "Using restricted loops in genetic programming for image classification," in *2007 IEEE Congress on Evolutionary Computation*, D. Srinivasan and L. Wang, Eds., IEEE Computational Intelligence Society. Singapore: IEEE Press, 25-28 Sep. 2007, pp. 4569–4576.

[3] J. R. Koza, D. Andre, F. H. Bennett III, and M. Keane, *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, Apr. 1999.

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

[5] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[6] K. E. Kinnear, Jr., "Evolving a sort: Lessons in genetic programming," in *Proceedings of the 1993 International Conference on Neural Networks*, vol. 2. San Francisco, USA: IEEE Press, 28 -1 1993, pp. 881–888.

[7] V. Ciesielski and X. Li, "Experiments with explicit for-loops in genetic programming," in *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*. Portland, Oregon: IEEE Press, 20-23 Jun. 2004, pp. 494–501.

[8] W. B. Langdon, "Data structures and genetic programming," in *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. Cambridge, MA, USA: MIT Press, 1996, pp. 395–414.

[9] N. Pillay, "Using genetic programming for the induction of novice procedural programming solution algorithms," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. Madrid, Spain: ACM Press, Mar. 2002, pp. 578–583.

[10] X. Li and V. Ciesielski, "Using loops in genetic programming for a two class binary image classification problem," in *AI 2004: Advances in Artificial Intelligence: Proceedings of the 17th Australian Joint Conference on Artificial Intelligence*, ser. Lecture Notes in Computer Science, G. I. Webb and X. Yu, Eds., vol. 3339. Cairns, Australia: Springer, Dec. 4-6 2004, pp. 898–909.

[11] G. Chen and M. Zhang, "Evolving while-loop structures in genetic programming for factorial and ant problems," in *AI 2005: Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence, Proceedings*, ser. Lecture Notes in Computer Science, S. Zhang and R. Jarvis, Eds., vol. 3809. Sydney, Australia: Springer, Dec. 5-9 2005, pp. 1079–1085.

[12] J. Cullen, "Evolutionary meta programming," in *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2009, pp. 81–88.