Experiments with Explicit For-loops in Genetic Programming

Vic Ciesielski
School of Computer Science
and Information Technology
RMIT University, GPO Box 2476V
Melbourne Victoria 3001
Email: vc@cs.rmit.edu.au

Xiang Li
School of Computer Science
and Information Technology
RMIT University, GPO Box 2476V
Melbourne Victoria 3001
Email: xiali@cs.rmit.edu.au

Abstract—**Evolving programs with explicit loops presents major difficulties, primarily due to the massive increase in the size of the search space. Fitness evaluation becomes computationally expensive and a method for dealing with infinite loops must be implemented. We have investigated ways of dealing with these problems by the evolution of for-loops of increasing semantic complexity. We have chosen two problems – a modified Santa Fe ant problem and a sorting problem – which have natural looping constructs in their solution and a solution without loops is not possible unless the tree depth is very large. We have shown that by controlling the complexity of the loop structures it is possible to evolve smaller and more understandable programs for these problems.**

I. Introduction

Loops are probably the most powerful constructs in programming. They provide a mechanism for repeated execution of a sequence of instructions. However, there is very little use of looping constructs in the programs evolved by genetic programming. There are a number of reasons for this. Firstly, loops are hard to evolve. It is necessary to evolve the start and end points and the body and to make sure they are consistent, for example, in some kinds of loops an index variable must appear in the body of the loop. Programs with loops generally take a lot longer to evaluate and some mechanism must be implemented for dealing with infinite loops. Secondly, it has turned out that there is a large class of useful problems which can be solved by evolving programs without loops. Thirdly, it is often possible to put the looping behaviour in the environment or into a terminal, for example, in the usual approach to the Santa Fe ant problem [1], the evolved program is repetitively invoked by the environment until some maximum number of steps has been exceeded while some robotics application might have a terminal such as 'go-forward-until-obstacle' [2].

At the most general level of abstraction there are two primary kinds of loops, the for-loop and the while-loop. In a for-loop the number of times the body of the loop is to be executed is known before execution of the loop begins. In a while-loop the number of repetitions is not known in advance and the loop body is repeated until some condition becomes true/false. In this paper we consider only for-loops.

A. Goals

Our goal is to investigate the evolution of programs with for-loops for problems which naturally involve some kind of repetitive behaviour.

We consider loops of the form

(FOR-LOOP1 NUM-ITERATIONS BODY)

in which BODY is executed NUM-ITERATIONS times and

(FOR-LOOP2 START END BODY)

in which a counter used in BODY is initialised to the value of START, BODY executed, the counter incremented and the process repeated until the value of the counter reaches END.

We investigate the following strategies for setting the values of NUM-ITERATIONS, START and END.

1) Set the value to a random type (Simple loop).
2) Set the value to the result of any computation permitted by the terminals and functions, including embedded looping constructs (Unrestricted loop).

Our expectation is that simple loops will be easier to evolve than unrestricted loops. We use these looping constructs on two problems which have natural repetitive characteristics, a modified Santa Fe ant problem in which we use FOR-LOOP1 constructs, and sorting of an array of numbers in which we use FOR-LOOP2 constructs. For each strategy we investigate whether the problem can be solved at all, the convergence behaviour, and the size of the evolved programs and compare the solutions with loops to solutions without loops.

II. Related work

There are very few reports in the literature on the use of loops in genetic programming. Koza [3, p135] described how to implement loops with automatically defined functions. He used the approach to solve the problem of computing the numerical average of LEN numbers in a vector V. In all of experiments described in his book, only a constrained form of automatically defined loops (ADL) is used and nested loops are not allowed. He used a pre-established maximum number of executions to ration the resources. He also used the ADL in a potential function set to solve an even-parity, a minimal sorting network and a robot controller problem. He concluded that ADLs could be a good factor for efficient solutions.

0-7803-8515-2/04/$20.00 ©2004 IEEE

Kinnear [4], [5] used an iterative operator with an index value to evolve a sorting algorithm. He restricted each loop to no more than 200 iterations and the total number of iterations in a program to 2,000. Considerable effort was expended in the design of the fitness function to encourage generality of the evolved solution. For example, he used tests of random sequences to obtain a high likelihood of generality. He added inverse size to the fitness measure and found that as well as decreasing the size, this improved generality.

Langdon [6] utilized a 'forwhile' construct to evolve a list data structure. In his experiments, nested loops were not allowed and the number of iterations was restricted to 32. The 'forwhile' provided the capability to process multiple list elements.

Wong and Leung [7] tried to evolve a general recursive solution for even-n-parity problems. They employed a logic grammar to enforce the base-case structure of recursion and regarded a program which did not produce a result after an allowed execution time as unfit.

Maxwell [8] developed a method to deal with infinite loops by incorporating time spent executing loops into a partial fitness calculation. This enables the evolution to use partial solutions where there are infinite loops, but which contain good building blocks. He used this method for the Santa Fe Ant Problem and found that it generated more efficient solutions than the optimum found in [9].

III. Syntax and Semantics of the For-Loops

We have used two variations of for-loops. For both variations we have experimented with simple loops and unrestricted loops. The syntax of the first variation is:

(FOR-LOOP1 NUM-ITERATIONS BODY)

and the semantics are quite straight forward, BODY is executed NUM-ITERATIONS times. During evolution, both NUM-ITERATIONS and BODY undergo crossover and mutation. In the case of simple loops NUM-ITERATIONS is restricted to a special integer type. The value is initially set to a random number between 1 and a programmer supplied value of MAX-ITERATIONS. During crossover and mutation typing is preserved so NUM-ITERATIONS can only be changed to another integer of this type. In the case of unrestricted loops, the value of NUM-ITERATIONS can be set by any function. This could involve the arithmetic functions {+, -} as well as several nested loops.

The syntax of the second for-loop variation is:

(FOR-LOOP2 START END BODY)

The semantics are also straight forward. BODY is executed once for each value of a counter between START and END. If START is greater than END, BODY is not executed. In the case of simple loops, START and END are restricted integer types as before, and in the case of unrestricted loops START and END can be the result of any possible computation, also as before.

In this implementation of looping, infinite loops are not possible, so no special actions are necessary in fitness evaluation.

IV. Evolution of the For-Loops

We use strongly typed genetic programming (STGP) [10] in our experiments. STGP simultaneously allows multiple data types and enforces closure by only generating parse trees which satisfy the type constraints. During genetic operations like crossover and mutation, only functions and terminals of the same type can be swapped or mutated.

In our function definitions for FOR-LOOP1 and FOR-LOOP2, {NUM-ITERATIONS, START, END} are of integer type. The for-loop function return type is of type dummy for simple-loops and integer for unrestricted-loops. During evolution, STGP will take care type matching and ensure only correct operations can be done.

V. Modified Santa Fe Ant Problem

The Santa Fe ant problem is described in detail in [9] and has been extensively studied [1]. The problem is to direct a robot ant to navigate through a twisting trail, the "Santa Fe Trail", on a 32 x 32 grid. There are 89 pieces of food on the trail. The robot eats the food when it enters into a square. The goal is for the robot to eat all of the food in as few moves as possible. The program can use three operations {Move, TurnRight, TurnLeft}. Move allows the robot to move one square forward. TurnRight and TurnLeft turns the robot to the right and left respectively. Each operation costs one step.

In previous work on the Santa Fe ant problem there has been no explicit iteration in the evolved programs. Iteration is accomplished implicitly in the environment by invoking the program as many times as necessary to eat all of the food or until some maximum number of steps (usually 600) has been expended. The fitness of a program is the number of pieces of food left after 600 steps. A successful solution is one in which all of the food has been consumed before 600 steps have elapsed. Three functions are used to glue the operations {IfFoodAhead, Prog2, Prog3} together. IfFoodAhead takes two arguments and executes one of its arguments depending on whether the square the robot is facing contains food or not. Prog2 and Prog3 take two or three arguments separately and execute them sequentially.

Our intention is to evolve programs in which there is no implicit looping. A program will be invoked only once, any looping behaviour must be explicitly in the program and the fitness of the program is the number of pieces of uneaten food after 600 steps. In our modified problem, the size of the grid is changed to 20x20 and 108 pieces of food are placed on the grid in 3 blocks of 6x6 as shown in figure 1. This regular placement of food is intended to encourage the evolution of loops within the evolved programs.

It is important to note that a solution to this problem which uses the nodes {Move, TurnLeft, TurnRight, IfFoodAhead, Prog2, Prog3} and has no explicit loop constructs will require a large tree. The optimal solution will require around 160 steps if the ant starts at position (0,0). A brute force solution which visits every square will need at least 400 hundred moves and 80 turns. A binary tree of depth 9 has this capacity. This is the reason we have restricted the grid size to 20x20.

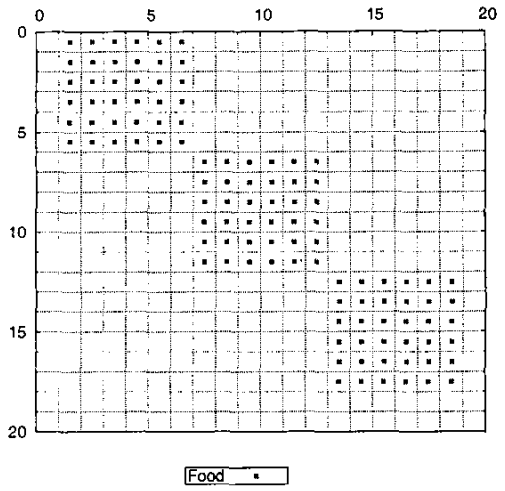

Fig. 1. Food Layout, modified ant problem

TABLE I

DEFINITION OF TERMINALS AND FUNCTIONS, MODIFIED ANT PROBLEM

Nodes Name	Description
Move::Terminal	The robot moves one square forward and it costs one step.
TurnLeft::Terminal	Turn the robot to the left direction of its current facing and cost one step.
TurnRight::Terminal	Turn the robot to the right direction of its current facing and cost one step.
RandTimes::Terminal	Generates a random integer between 0-6 or 0-20 or 0-50.
IfFoodAhead::Function	Takes 2 arguments and executes the first argument if there is a food in front, else executes the second.
Prog2::Function	Takes 2 arguments and executes them sequentially.
Prog3::Function	Takes 3 arguments and executes them sequentially.
For-Loop1::Function	Takes 2 arguments. The first argument indicates number of times the second argument is executed. It returns number pieces of food left after the execution of the loop body.

A. Experiments

All experiments have been run with the functions and terminals shown in table I. The values of other GP variables are shown in table II.

B. Experimental Results

Figure 2 shows the fitness of the best individual, averaged over 100 runs, for 2000 generations of evolution for the case where MAX-ITERATIONS was 6. These results were somewhat surprising. Since the a large tree is necessary to solve the problem without loops, as described above, we expected that programs with loops might perform better. However, we expected that the simple loops would be easier to evolve than the unrestricted ones. As figure 2 reveals, the opposite was the case. The reason for this is not clear. We think that in some way the unrestricted loops constrain the search more than the by using subtle feedback about the quality of the solution.

TABLE II

VARIABLE SETTINGS, MODIFIED ANT PROBLEM

Variable Name	Value
Population Size	100
Mutation Rate	0.28
Crossover Rate	0.70
Elitism Rate	0.02
Maximum Depth	8
Minimum Depth	1
Termination Criteria	2000 generations or all food (108 pieces) is found or 600 steps are reached.

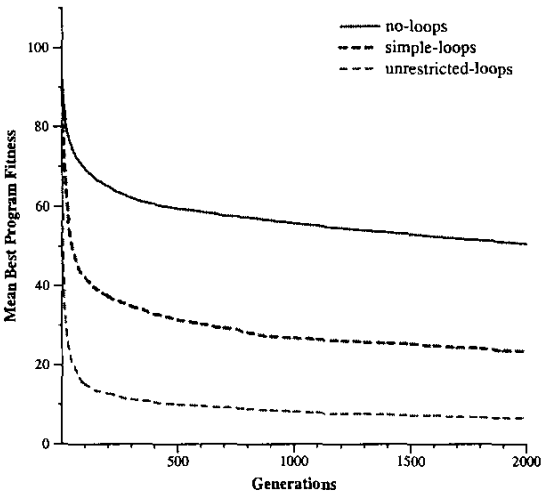

Fig. 2. Mean best program fitness, modified ant problem, max-iterations=6, average of 100 runs

Figure 3 shows the cumulative probability of getting a successful solution, that is, the evolved ant eats all of the food, corresponding to the fitness values given in figure 2. None of the runs without loops gave a successful solution. At 2000 generations, 12 of the 100 simple loop and 23 of the unrestricted loop runs gave a solution.

Figure 4 shows a comparision of the fitness of the best individual for different choices of MAX-ITERATIONS for simple loops. The figure shows that higher values of MAX-ITERATIONS lead to better programs. There is, however, an unfortunate side effect that is not evident from the figure – the execution time rises dramatically. The 100 runs for MAX-ITERATIONS of 6, 20 and 50 took 1 hour, 3 hours and 1 day, respectively on our hardware. A similar analysis for unrestricted loops showed no difference for the same values of MAX-ITERATIONS.

Since programs with a small number of loops are usually more understandable, we performed a number of runs in which the fitness function was modified to favour programs with fewer occurrences of FOR-LOOP1. This was done by counting the number of occurrences of FOR-LOOP1 in the text of the program and adding it to the number of pieces of food left after program execution. Thus, if two programs consume the same amount of food, the one with fewer loops will be

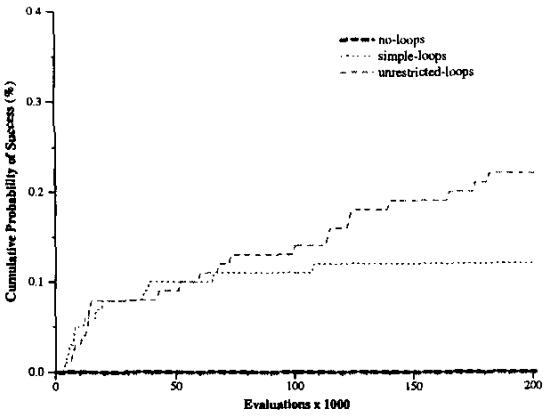

Fig. 3. Cumulative probability of success, modified ant problem, max-iterations=6, average of 100 runs, (the no-loops line is on the x axis)

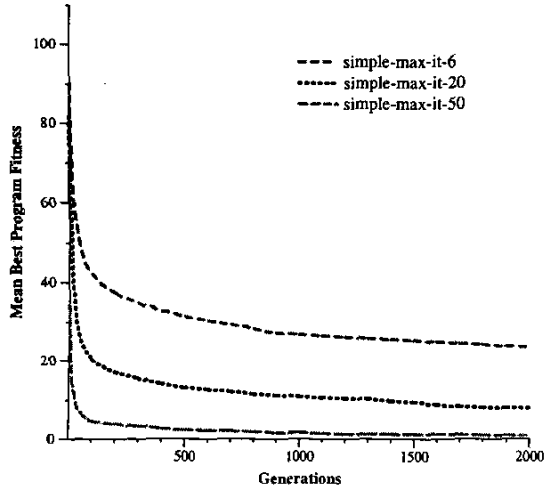

Fig. 4. Mean best program fitness, different values of MAX-ITERATIONS, average of 100 runs, modified ant problem

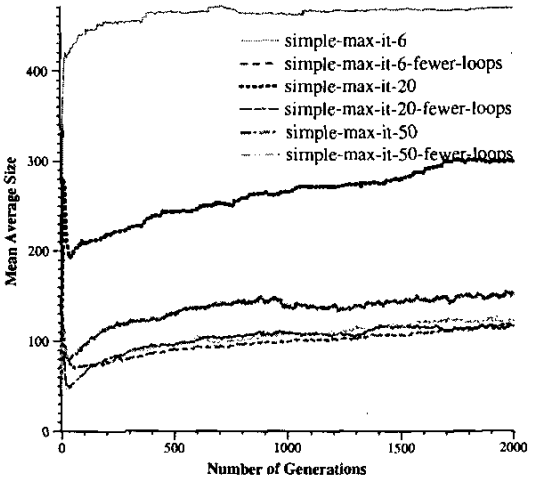

Fig. 5. Favouring programs with fewer loops, best fitness, averages of 100 runs, modified ant problem

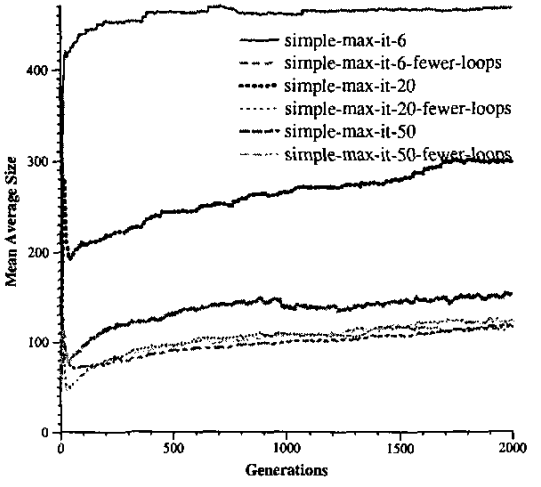

Fig. 6. Favouring programs with fewer loops, program size, averages of 100 runs, modified ant problem

fitter. Figure 5 shows a comparison of best fitness over the generations while figure 6 shows a comparison of program size. As can be seen from figure 5, favouring programs with fewer loops has a dramatic effect on fitness for simple loops but has no effect on the unrestricted loops. Figure 6 reveals quite a difference in program size if fewer loops are favoured. All but one of the curves shows an initial drop in program size. We believe that reason for this is the following: The programs in the initial population are generated by the ramped half-and-half method. Larger programs are highly likely to have more occurrences of loops. In fitness evaluation programs are terminated after executing 600 steps. Large programs will use up their allocation of steps before consuming much of the food and hence will not be as fit as the smaller programs. These unfit programs are not selected for mating and hence are removed from the next generation. Eventually these smaller programs increase in size as their fitness improves.

C. *Analysis of Solutions*

When MAX-ITERATIONS was large (20,50) the evolved solutions traversed every square in the grid. A typical pattern is shown in figure 7.

Solutions favouring a smaller number of loops tended to have larger loop bodies, shorter depth and size, and to be more understandable. An example of such a solution is shown in table III. This solution, whose traversal pattern is shown in figure 8, was found at generation 294 using the strategy of favouring programs with fewer loops. It uses 168 steps to eat the food and is close to optimal. The robot moves in a zigzag manner, switching its head left and right to detect food. If there is food ahead, it moves ahead and turns back by executing two TurnRight actions. If not, it turns left. Depending on the result of sensing, the robot either does 2 forward moves or just one move and then senses again.

TABLE III

A SOLUTION EVOLVED BY FAVORING PROGRAMS WITH FEWER LOOPS, MODIFIED ANT PROBLEM

```
SIMPLIFIED : (ForLoop1 times5
  (ForLoop1 times5
    (ForLoop1 times5
      (IfFoodAhead
      ((Prog2 move turnRight) turnRight)
      (Prog2 turnLeft (IfFoodAhead (Prog2 move move) move))))))
```

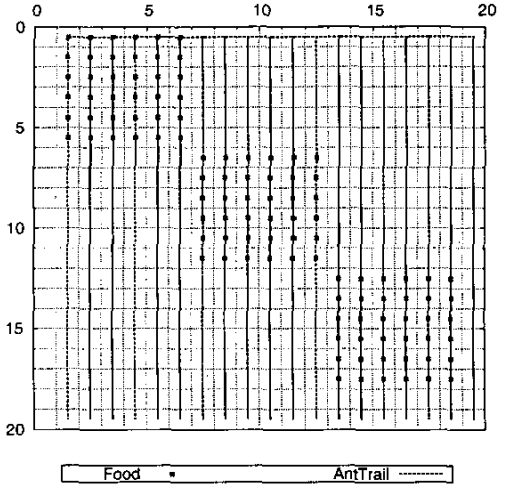

Fig. 7. Traversal pattern for a solution evolved with MAX-ITERATIONS=20, modified ant problem

In contrast, table IV shows the smallest successful solution evolved when there as no favouring of programs containing fewer loops. This was generated for a MAX-ITERATIONS of 6. As can be seen from table IV it has more nodes and fragments and it is harder to understand what the program is doing by analysing the code. Figure 9 shows the corresponding traversal pattern.

It is very hard to get a solution with the non-loop approach. Out of three hundred runs with a maximum depth of 10, only one found an answer. The solution is enormous. It contains more than 5000 nodes and is impossible to understand. It takes 4 full A4 pages to print out. Figure 10 shows the traversal path of the solution. The robot uses 1704 steps to complete the task. As mentioned earlier, no solution was found at a maximum depth of 8.

TABLE IV

A SOLUTION EVOLVED WITHOUT FAVOURING PROGRAMS WITH FEWER LOOPS, MODIFIED ANT PROBLEM

```
(ForLoop1 times5 (ForLoop1 times4
(Prog3 (ForLoop1 times4 move)
(Prog3 (IfFoodAhead (Prog3 move move move) move)
(Prog3 (Prog2 turnRight move) turnRight (IfFoodAhead
(ForLoop1 times4 move) (Prog2 turnRight turnLeft)))
(IfFoodAhead (Prog3 move move move) move)) (Prog3
(IfFoodAhead (IfFoodAhead (Prog3 turnLeft turnRight
move) move) turnLeft) turnLeft (Prog2 move move)))))
```

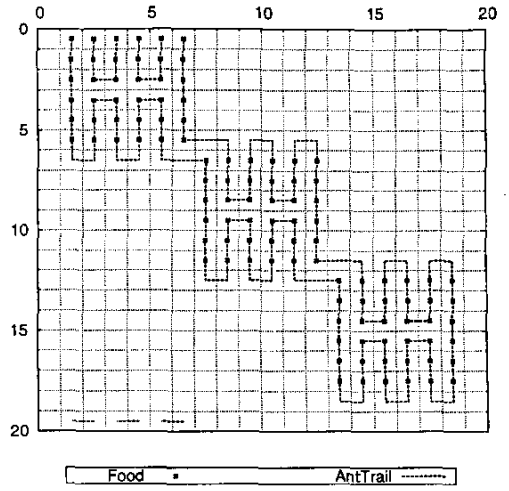

Fig. 8. Traversal pattern of the program shown in table III

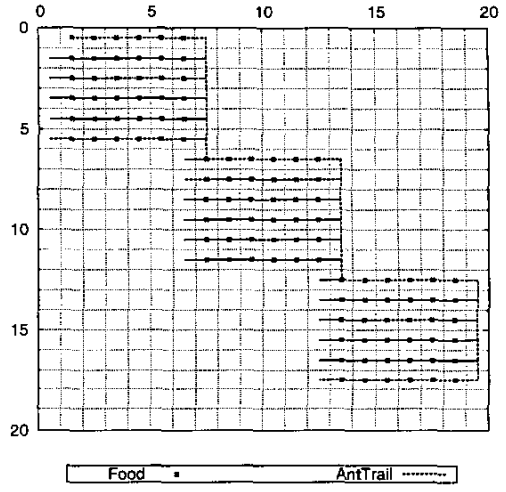

Fig. 9. Traversal pattern of the program shown in table IV

VI. Sorting Problem

Sorting an array of numbers is another problem that has natural looping characteristics. The sorting algorithms taught in introductory computer science classes, such as selection sort and bubble sort, require two nested loops. The two basic operations are comparing and swapping. Evolution of sorting programs is not well suited to genetic programming because of difficulties with fitness evaluation. It is very difficult to develop a tractable fitness function that guarantees that any array of arbitrary length will be sorted after the evolved program has been executed.

Genetic programming solutions to sorting problems have been studied by Kinnear [4], [5] who attempted to evolve generalised sorting algorithms and Koza [11, p335] who

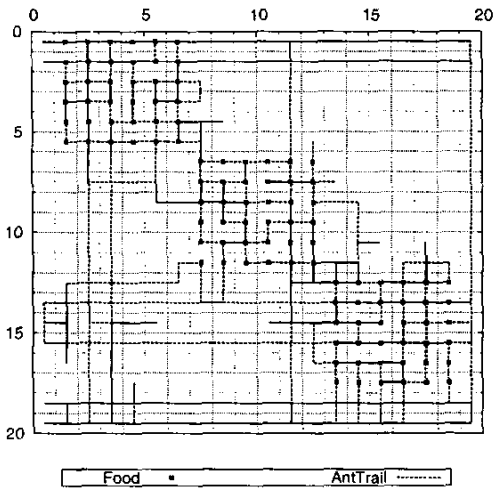

Fig. 10. Traversal pattern of the only solution evolved by the non-loop method, max depth=10, modified ant problem

TABLE V

DEFINITION OF TERMINALS AND FUNCTIONS, SORTING PROBLEM

Nodes Name	Description
POS::Terminal	Random number in range 0..6
IfLessThanSwap::Function	Takes two arguments. If arg1 is less than arg2 the positions are swapped and the position of the larger value is returned
Prog2::Function	Takes 2 arguments and executes them sequentially.
ForLoop2::Function	Takes 3 arguments, start position, end position and body.
$+, -, \times, /$	with usual meanings

attempted to minimise the number of comparisons.

Our focus is on the evolution of loops of different complexities and on the comparison of loop and non loop solutions. We do not address the issue of a generalised fitness function. Fitness is evaluated by applying an evolved program to all 7! = 5040 permutations of arrays of length 7 and counting the number of out-of-place elements. The actual fitness calculation is shown in table VII. Seven was chosen as the upper limit of array size so that the runs could be done in reasonable time.

A. Experiments

As before we have carried out runs with no loops, with simple loops where START and END are restricted to an integer type and with unrestricted loops where START and END can be set by any possible calculation. The terminals and functions used are shown in table V. Values of the other GP variables are shown in table VI.

B. Results

The fitness of the best individual for all methods is shown in figure 11. The corresponding cumulative probability of success is shown in figure 12. Programs with loops are clearly fitter and, in fact, for both kinds of loops all runs found a solution within 40 generations. In contrast, at 100 generations only 34 of the 50 runs without loops had found a solution.

There seems to be an inconsistency between figures 11 and 12. The runs of simple-loops have the best possible mean fitness from the first generation and unrestricted-loops take several generations to get there. Yet the cumulative probability of success rises faster for unrestricted ones. This is because the Y-axis scale for the mean best program fitness is huge. Best programs with simple loops get very good mean best fitness, but none of them actually reaches a solution in the first several generations. Because of the scale, the fitness diagram looks like the simple approach gets to the solutions quicker.

The size of the best individual is shown in figure 13. Surprisingly the programs with loops are bigger than the programs without loops. This is because at an array size of 7, the programs without loops are still relatively small and the benefits of loops are not yet apparent. As the size of the array grows larger the non loop solution must also grow, perhaps exponentially. Some preliminary work that we have done on an array size of 11 has led to similar results as with the ant problem, that is, the programs without loops were huge and the cumulative probability of success very small, while the programs with loops were smaller and the cumulative probability of success considerably higher.

The number of comparisons made by the best individual is shown in figure 14. The programs with loops are making more comparisons. This is related to program size.

Table VIII shows one of the best evolved individuals without loops in terms of number of comparisons and number of swaps. Table IX shows one of the best programs evolved with simple loops. Analysis of this program reveals a general strategy of moving large elements to one end, while the non loop program is very difficult to understand.

TABLE VI

PARAMETER VALUES, SORTING PROBLEM

Variable Name	Value
Population Size	100
Mutation Rate	0.28
Crossover Rate	0.70
Elitism Rate	0.02
Maximum Depth	7
Minimum Depth	1
Termination Criteria	100 generations elapsed or the array is sorted

TABLE VII

ALGORITHM FOR FITNESS CALCULATION, SORTING PROBLEM

```
int calculateFitness(int _length, int *_array)
{
  int i, result = 0;

  for( i=1; i <= _length; i++)
  {
    result += abs( _array[i-1] - i );
  }
  return result;
}
```

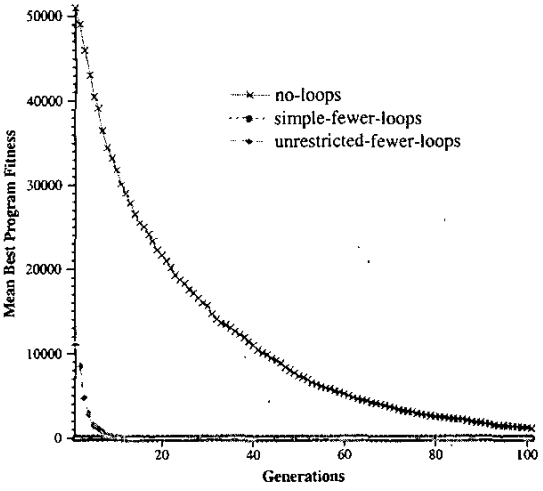

Fig. 11. Mean best program fi tness comparison, Averages of 50 runs, sorting problem

TABLE VIII
ONE OF THE BEST PROGRAMS EVOLVED WITHOUT LOOPS, 18 COMPARISONS AND 8 SWAPS (ILETS = IFLESSTHANSWAP)

```
(Prog2 (Prog2 (Prog2 (Prog2
   (ILETs POS0 POS2) (ILETs POS5 POS1))
   (ILETs POS4 POS6))
(Prog2 (Prog2
   (ILETs POS1 POS6) (ILETs POS0 POS4))
(Prog2 (Prog2
   (ILETs POS4 POS5) (ILETs POS0 POS4))
   (ILETs POS3 POS4))))
(Prog2 (Prog2 (Prog2
   (ILETs POS1 POS3) (ILETs POS2 POS4))
(Prog2 (Prog2
   (ILETs POS4 POS6) (ILETs POS5 POS4))
(Prog2
   (ILETs POS1 POS2) (ILETs POS2 POS5))))
(Prog2 (Prog2
   (ILETs POS0 POS1) (ILETs POS3 POS5))
(Prog2 (Prog2 (Prog2 Dummy Dummy)
   (ILETs POS2 POS3)) (ILETs POS4 POS5)))))
```

The results on the sorting problem are not as good as those on the ant problem. The main reason for this is that the sorting problem is considerably harder. It is known that sorting can be done with two nested loops, however, the limits of the inner loop need to be co-ordinated with the loop index of the outer loop. In our formulation of the problem this could only happen by random chance. This did not occur in any of the runs and the evolved programs contained large numbers of uncoordinated loops.

Comparisons of the efficiency of the different approaches, as well as comparisons with standard sorting algorithms are shown in table X. For each row of the table the given algorithm was applied to all of the 5040 test cases and the number of comparisons and the number of swaps were counted. The numbers given are the average number of swaps per test case. The evolved programs are competitive with conventional algorithms.

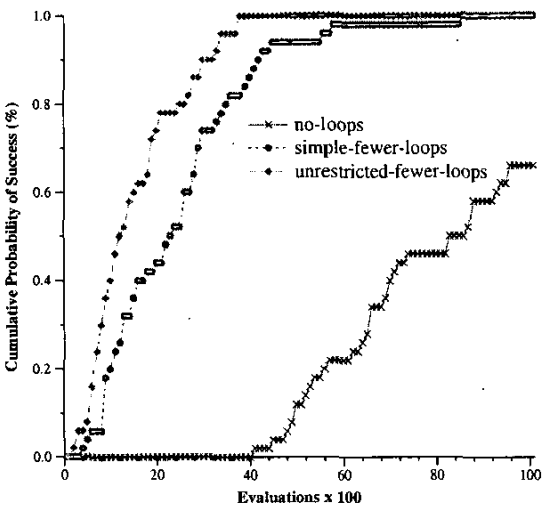

Fig. 12. Cumulative probability of success, sorting problem

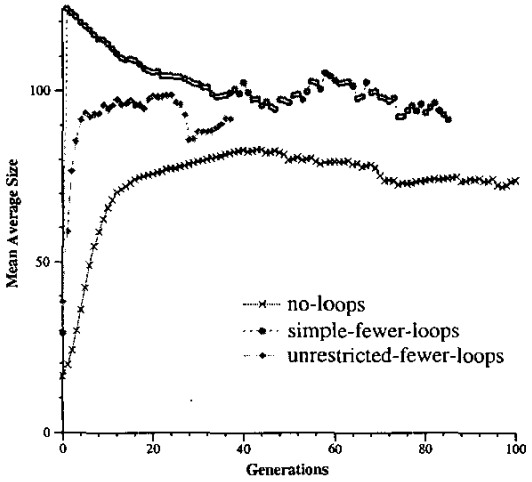

Fig. 13. Size of the best individual, Averages of 50 runs, sorting problem

TABLE IX
A GOOD PROGRAM WITH SIMPLE LOOPS, 22 COMPARISONS AND 10 SWAPS, SORTING PROBLEM

```
(Prog2 (Prog2 (Prog2
   (ForLoop2 POS3 POS4 (ILETs i (i+1)))
   (ForLoop2 POS2 POS6 (ILETs i (i+1))))
(Prog2
   (ForLoop2 POS3 POS4 (ILETs i (i+1)))
   (ForLoop2 POS4 POS5 (ILETs i (i+1)))))
(Prog2
   (ForLoop2 POS1 POS6 (ILETs i (i+1))))
(Prog2 (Prog2
   (ForLoop2 POS3 POS2 (ILETs i (i+1)))
   (ForLoop2 POS1 POS3 (ILETs i (i+1))))
(Prog2
   (ForLoop2 POS1 POS3 (ILETs i (i+1)))
   (ForLoop2 POS0 POS6 (ILETs i (i+1)))))))
```

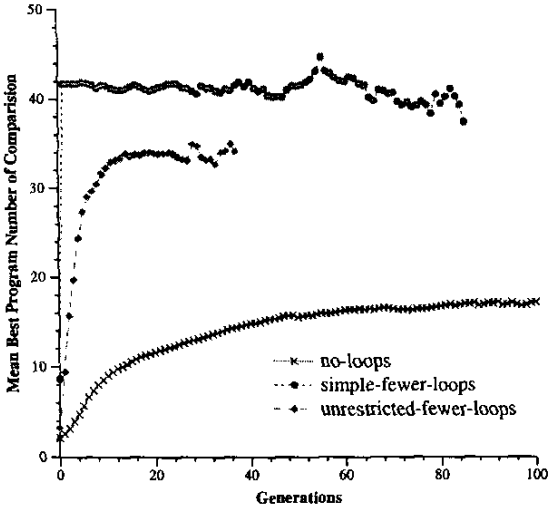

Fig. 14. Number of comparisons made by best individual, Averages of 50 runs, sorting problem

TABLE X
DIFFERENT SORTING METHODS FOR 7 ELEMENT ARRAYS, 5040 TEST CASES, SORTING PROBLEM

Methods	Comparisons	Swaps
Bubble Sort	21.00	10.50
Shell Sort	16.50	16.50
Insertion Sort	17.09	15.50
Selection Sort	21.00	6.00
Quick Sort	44.42	10.19
No loops	17.00	7.33
Simple loops	22.00	10.00
Unrestricted loops	21.00	9.00

VII. CONCLUSIONS

Our goal was to evolve programs with loops to solve problems with some naturally occurring repetitive behaviour. By restricting the semantic complexity of for-loops we have succeeded in evolving small, efficient and reasonably understandable solutions to a modified Santa Fe ant problem and to a sorting problem. In the case of the ant problem, a solution without loops contained over 5,000 nodes and many of the evolved solutions with loops had fewer than 30 nodes.

Surprisingly, restricting the loop constructs to simple loops in which the number of iterations or the start and end values of a loop were a special integer type, was not as effective as unrestricted loops. The unrestricted loops generally evolved fitter solutions in fewer generations. The reason for this requires further investigation.

Using the fitness function to favour programs with fewer loops was very beneficial. The programs evolved in this way were smaller and more understandable and generally fitter than programs evolved without this bias.

Most researchers and practitioners in genetic programming have tended to avoid the use of looping constructs, primarily due to difficulties in evolving consistent programs and dealing with infinite loops. In formulating the functions for a problem domain, looping constructs are rejected out of hand. Our results suggest that looping constructs are worth considering when the problem domain has some repetitive charateristics. While evolution of generalised loops is currently not possible, looping constructs with carefully designed syntax and semantics can be used to great advantage.

ACKNOWLEDGMENT

This work was partially supported by grant EPPNRM054 from the Victorian Partnership for Advanced Computing.

REFERENCES

[1] W. B. Langdon and R. Poli. Why ants are hard. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[2] V. Ciesielski, D. Mawhinney, and P. Wilson. Genetic programming for robot soccer. In *Proceedings of the RoboCup 2001 International Symposium*, pages 319–324, Seatte, USA, July 2002. Springer.

[3] John R. Koza, Forrest H Bennet III, David Andre, and Martin A. Keane. *Genetic Programming III; Darwinian invention and problem solving*. Morgan Kaufmann, 1999.

[4] Kenneth E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA, 28 -1 1993. IEEE Press.

[5] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, University of Illinois at Urbana-Champaign, 17-21 1993. Morgan Kaufmann.

[6] William B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[7] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[8] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1998 United Kingdom Automatic Control Council International Conference on Control (UKACC International Conference on Control '98), University of Wales*, volume 455, Swansea, UK, 1-4 1998. IEEE Press.

[9] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[10] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[11] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.