Evolving Data Structures with Genetic Programming

William B. Langdon
Computer Science Dept.
University College London,
Gower Street, London, WC1E 6BT, UK
Email W.Langdon@cs.ucl.ac.uk

Abstract

Genetic programming (GP) is a subclass of genetic algorithms (GAs), in which evolving programs are directly represented in the chromosome as trees. Recently it has been shown that programs which explicitly use directly addressable memory can be generated using GP.

It is established good software engineering practice to ensure that programs use memory via abstract data structures such as stacks, queues and lists. These provide an interface between the program and memory, freeing the program of memory management details which are left to the data structures to implement. The main result presented herein is that GP can automatically generate stacks and queues.

Typically abstract data structures support multiple operations, such as put and get. We show that GP can simultaneously evolve all the operations of a data structure by implementing each such operation with its own independent program tree. That is, the chromosome consists of a fixed number of independent program trees. Moreover, crossover only mixes genetic material of program trees that implement the same operation. Program trees interact with each other only via shared memory and shared "Automatically Defined Functions" (ADFs).

ADFs, "pass by reference" when calling them, Pareto selection, "good software engineering practice" and partitioning the genetic population into "demes" where also investigated whilst evolving the queue in order to improve the GP solutions.

1 INTRODUCTION

Recent work by Teller [Tel94a] shows genetic programming can automatically create programs which explicitly use memory. He has shown that inclusion of *read* and *write* primitives can make the GP language *Turing complete*, i.e. any conventional program can be written in the language [Tel94b]. However it is still an open problem as to which subclass of programs can be effectively evolved.

Human programmers have long recognised, that in addition to Turing completeness, programming languages should encourage programs to be structured. In particular, program production, maintenance and testing are eased if the software is written so that it be independent of memory access implementation details. This is achieved by using abstract data structures, such as stacks, queues and lists, in order to provide an interface between programs and memory.

We anticipate that if evolutionary computation is to solve many difficult problems it must adopt a structured approach, particularly in its use of memory. We demonstrate how GP can automatically generate two abstract data structures, stacks and queues.

The GA we use is based on Koza's GP [Koz92], but each individual within the population comprises several program trees, one for each operation (see Figures 2 and 3). These trees are independent, with crossover occurring only between like trees. Each interacts with the others via shared memory or shared "Automatically Defined Functions" ADF [Koz94]. We show it is possible for this GA to simultaneously evolve multiple co-operating but independent functions.

In the following sections we describe our experiments which show that both an integer stack (Section 2) and a First-In First-Out (FIFO) integer queue (Section 3) can be evolved. Section 4 discusses the results achieved and possible further work.

"Pass by reference" (Section 3.7.1) is introduced to GP in order to facilitate the evolution of primitives which update variables. Pareto selection (Section 3.7.2) is introduced as it provides a natural way of comparing programs which perform multiple operations. We use "good software engineering practice" within the fitness function and syntax of the GP language to guide the GA's search (Section 3.7.3). Partitioning the genetic population into "demes" appears to mitigate against premature convergence (Section 3.7.4).

2 EVOLVING A STACK

2.1 PROBLEM STATEMENT

Our definition of a stack is given in Table 1. Whilst based upon Aho et al [AHU87], it has been simplified by removing the checks for stack underflow or overflow and "pop" returns the current top of the stack as well as removing it. Our problem only requires the solution to implement a stack of ten integers, however the programs evolved scale up to stacks of any depth. In fact the fitness function tests only as far as depths of four items.

2.2 ARCHITECTURE

Each individual within the population is composed of five trees, each of which implements a trial solution to one of the five operations that form the complete stack program (see Figure 1). Using this architecture it is possible to evolve all five operations simultaneously from randomly generated program trees.

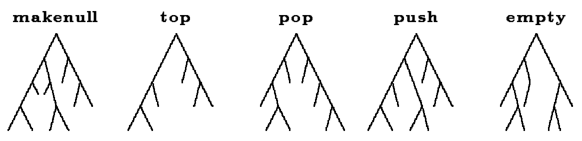

Figure 1: One Individual – Five Trees

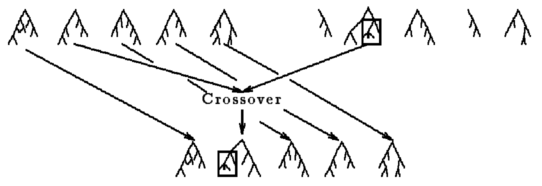

Figure 2: Crossover in One Tree at a Time

This multiple tree architecture was chosen so that each tree contains code which has evolved to implement a single operation. It was felt that this would ease the formation of "building blocks" of useful functionality and enable crossover, and other genetic operations, to assemble working implementations of the five operations from them. Consequently, complete stack programs could be formed whilst each of its trees improved.

Each new individual is created either by copying all five trees of the parent program (10%) or via crossover between two parent programs (90%). When crossing over, one type of tree is selected at random. The trees of the other types are copied without modification from the first parent to the offspring. The remaining tree is created by crossover between the trees of the chosen type in each parent in the normal GP way [Koz92]. The new tree has the same root as the first parent. Each mating produces a single offspring, most of whose genetic material comes from only one of its parents. Crossover is limited to a single tree at a time in the expectation that this will reduce the extent to which it disrupts "building blocks" of useful code. Crossing like trees with like trees is similar to the crossover operator used by Koza in most of his experiments involving ADFs [Koz94].

GP creates each tree from terminals (leafs) and functions (branch nodes). Collectively these are called primitives. Initially, for simplicity, the same primitives were available to each tree. However "good software engineering practice" suggests a number of rules that might help the GP. For example, empty should not have side effects and therefore primitives with side effects (e.g. write) should not occur in empty's tree. From Section 3.6.2 onwards rules of this type are used for the queue problem.

2.3 TERMINALS AND FUNCTIONS USED TO EVOLVE STACK

We used primitives such as those a human programmer might use. The terminals chosen were the constants 0, 1, max (10) and arg1, the variable aux and Inc_Aux and Dec_Aux: a) max denotes the maximum size of the stack, b) arg1 holds the input for push; it is zero if used other than by push, c) aux is intended to be used as a pointer by holding memory addresses, however the GP may use it as it pleases, and d) Inc_Aux and Dec_Aux update aux by ± 1 and return its new value.

The functions chosen were +, −, Write_Aux, read and write: a) Write_Aux sets aux to the value of its argument but returns aux's previous value, and b) read and write functions read the value in and/or update the value stored in one of 63 integer memory cells. They are based on Teller's [Tel94a], however access to memory outside the range -31...31 aborts the program. An aborted program fails the current test and is not tested further but keeps its current score. In some queue experiments (Section 3.6.2 onwards) programs continue despite memory address errors (in which case read and write return zero) and in Section 3.7.5 only 31 memory cells were used.

```
read(a):  If a valid memory index then
            read := store[a];
          Else
            Abort program;
```

Table 1: Pseudo Code Definition of the Five Stack Operations

makenull	sp	:= maxlength + 1;	initialise stack
empty	empty	:= (sp > maxlength);	is stack empty or not?
top	top	:= stack[sp];	top of the stack
pop	pop	:= stack[sp];	return top of stack
	sp	:= sp + 1;	and remove it
push(x)	sp	:= sp − 1;	place x on top of stack
	stack[sp]	:= x;	

```
write(a,x):  If a valid memory index then
               write := store[a];
               store[a] := x;
             Else
               Abort program;
```

2.4 FITNESS FUNCTION

The fitness of each individual program is the number of tests it passes (i.e. returns the correct answer) when each of its constituent operations are called in a series of four fixed test sequences, each containing 40 calls. These were chosen to test correct operation of stack programs, up to a depth of four. Each sequence starts with makenull, never causes stack overflow or underflow and top is never called when the stack is empty. The 47 values pushed on to the stack were also fixed. They were selected at random from the range -1000 and 999.

The answers returned by makenull and push are ignored; they are tested indirectly by seeing if the other operations work correctly when called after them. They are both scored as if they had returned the correct result. The integer value returned by empty is converted to a boolean by treating all values > 0 as true.

All storage, i.e. the indexed memory and aux, is initialized to zero before each test sequence is started. Nb. no information about the program's internal behaviour is used.

2.5 PARAMETERS

The parameters used where those those established by Koza [Koz94, page 655] except GP-QUICK [Sin94] uses: a steady state GA; a single offspring per crossover. The default tournament size of 4 appears to give sufficient selection pressure, however the default program size limit was increased five fold, to 250, in order to allow ready growth of all five trees. A population of 1,000 proved sufficient for the stack.

2.6 RESULTS

With the above parameters in a group of 60 runs, four produced correct programs. In three the stack grows down memory and in the other it grows up; all use memory cell zero first. Whilst each program is different, examination shows they all correctly implement a general stack, i.e. a stack of any depth rather than just ten (subject to the available memory). Figure 3 shows the simplest correct program, its essential code is shown within the boxes.

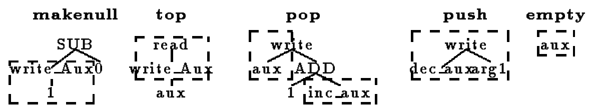

Figure 3: Evolved Stack Program (1)

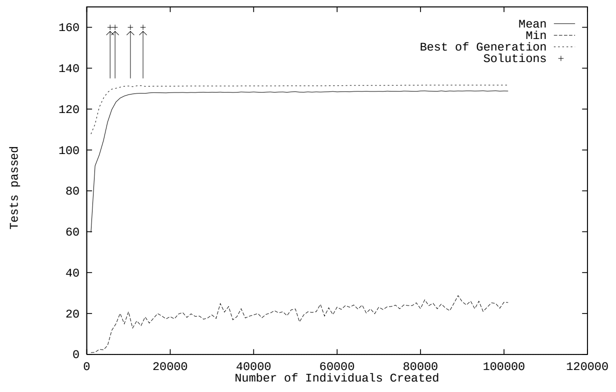

Figure 4: No. Trial Stacks Created v. Fitness, Means of 60 runs

Once the runs have completed it is possible to estimate, using Figure 4, the probability of a stack being evolve at generation i when using a population of size M, $P(M, i)$. From $P(M, i)$ the number of runs required to obtain at least one stack can be calculated. Using the formula in [Koz92, page 194], estimating $P(1000, 14)$ at 4/60 (i.e. 4 successes in 60 trials) and requiring the chance of not finding any stacks to be less than 1% gives 67 runs. I.e. 67 independent runs, each running for up to 14 generations, will ensure that the chance of producing at least one stack is better than 99%. This would require a total of up to $14 \times 1,000 \times 67 = 938,000$ trial programs to be tested.

For the sake of comparison, a large number of random programs were generated, using the same mechanisms as the GP, and tested against the same fitness tests. A

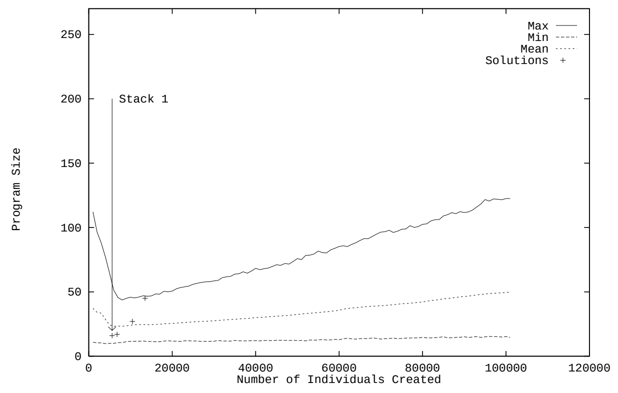

Figure 5: No. Trial Stacks Created v. Program Size, Means of 60 runs

total of 49,000,000 randomly produced programs were tested, none passed all of the tests.

3 EVOLVING A QUEUE

This Section describes a series of experiments aimed at repeating the success with evolving a stack, but this time evolving a FIFO queue. The design choices for the queue, were based on those used with the stack. These produced initial partial solutions (Section 3.6), which were followed by design changes. Section 3.6.2 shows that queues can readily be evolved if powerful primitives are available. Section 3.7 shows that such primitives need not be given but can be evolved by the GP whilst it solves the queue problem, however considerably more trial solutions needed to be tested.

3.1 PROBLEM STATEMENT

Our definition of a queue is again based upon that given by Aho et al [AHU87]. As with the stack there are five operations: makenull, front, dequeue, enqueue and empty. We simplify the problem as before, so the queue problem is very close to the stack with the replacement of front for top, dequeue for pop and enqueue for push.

As Aho et al show, implementing a queue as a circular buffer can be done by allowing a gap between the oldest and newest items in the queue. This gap takes at least one cell, therefore our problem restricts the number of items in the queue to nine, rather than ten used for the stack. As we will show it is possible to evolve programs that will scale up to any queue length. However GP also evolved programs specific to the length of queue and so care had to be taken in the fitness function to test the full range of queue lengths.

3.2 ARCHITECTURE

The five trees used with the stack were augmented by a sixth: adf1, an ADF which may be called from any tree (to avoid infinite loops, recursive calls abort the program). In later experiments (Section 3.6.2 onwards) dequeue may call front and in the final experiments (Section 3.7) the ADF concept was extended to allow ADFs to modify their arguments by using a form of passing data by reference.

3.3 TERMINALS AND FUNCTIONS USED TO EVOLVE QUEUE

The details of terminals and functions are given in Tables 2 and 3. They are essentially those for the stack problem except: a) two auxiliary variables, aux1 and aux2, rather than one, b) Write_Aux$n$ replaced by Set_Aux$n$, which yield the new value of aux$n$ rather than its original value, and adding c) adf1 and d) mod. mod is a "protected" modulus operator: mod(a,b) = remainder of a/|b|, unless b = 0 when mod returns a.

In later experiments (Section 3.6.2 onwards) we restrict the primitives that can be used in which trees and add the two argument functions PROG2 and QROG2. a) PROG2 yields the value of its second argument and b) QROG2 that of its first. They were added so that they could be used to link together subtrees without transforming their values as other binary functions do (e.g. +).

3.4 FITNESS FUNCTION

The fitness function is based upon that used with the stack. Test sequences always start with makenull, never enqueue more than nine items, and never call dequeue or front on an empty queue. All storage is initialized to zero before each test sequence.

Initially the test sequences were identical to that used for the stack, with the replacement of enqueue for push and dequeue for pop. Also, only when empty returns zero, is its answer treated as true. However the GP was unable to generalize from these limited tests and programs evolved which passed all the tests but did not correctly implement a queue. As these were produced the fitness function was changed.

After the discovery of memory hungry solutions (Section 3.6) the test sequences where changed by adding a long sequence (160 tests) and to ensure the whole range of queue lengths were tested. The fifth test sequence contains makenull only once and so ensures memory hungry solutions are penalized by exhausting the available memory.

Initially the fitness function was the same as the stack, except makenull and enqueue test passes where reduced in weight by dividing them by 20.0 before adding them to the other scores. This was modified to reward "good software engineering practice". Initially (Section 3.6.1) by subtracting 2.0 per memory cell above 15 used. 2.0 was chosen to ensure caterpillar like programs which use more memory and so pass more tests actually have lower fitness. Later (Section 3.6.2 onwards) the single fitness value was replaced with Pareto scoring (Section 3.7.2) and excessive memory usage was a factor in the Pareto fitness.

With a uniform distribution of data values, partial solutions were produced which exploited the fact that the value zero was never enqueued. From Section 3.6.1, a tangent distribution was used to bias the distribution of test data towards zero, so that it contains small values, like zero, but still contains large values. With a tangent distribution approximately 50% of values lie in -F...F. The scaling factor, F, was initially 31.4, which covers the range of legal memory address, but it was progressively reduced.

3.5 PARAMETERS

The parameters used are as for the stack except, the population size was increased to 10,000 and the last series of experiments split the population into demes (3.7.4).

3.6 INITIAL RESULTS

A number of runs were made which yielded partial solutions to the queue problem. One group of these are known as "caterpillars" (Figure 6). The two auxiliary registers are used as pointers to the queue's head and tail and are incremented by each enqueue or dequeue. However they are not reset so the queue crawls its way across memory, like a caterpillar. Except for requiring indefinite amount of memory, entirely correct solutions were automatically evolved.

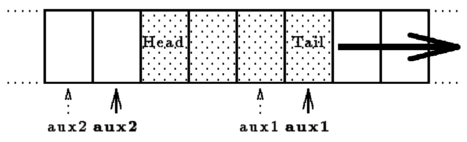

Figure 6: Queue works up memory like a Caterpillar. Nb. data does not move.

3.6.1 Shuffler

In a group of 379 runs one solution which passes all 320 tests was found. This is known as the "Shuffler" (Figures 7 and 8). Several more solutions of this type have been found in runs with slightly different parameters or primitives. Many partial solutions of the shuffler type have also been found.

As Figure 7 shows, this solution correctly implements a FIFO queue of up to nine items. Unexpectedly it does this by moving the contents of the memory cells. I.e. as each item is removed from the queue, all the remaining items are moved (or shuffled) one place down. Thus the front of the queue is always stored in a particular location. One of the auxiliary variables is used to denote the newest item in the queue. The other variable is used by dequeue as a temporary pointer.

Figure 8 gives the impression that dequeue was built up of code fragments, write(Inc_Aux2,). This program seems to have evolved because as crossover inserted another write(Inc_Aux2,) into dequeue, the whole program was able to process longer queues and so pass more tests. I.e. its fitness increased and so the proportion of write(Inc_Aux2,) code fragments increased in the population making further similar crossovers more likely.

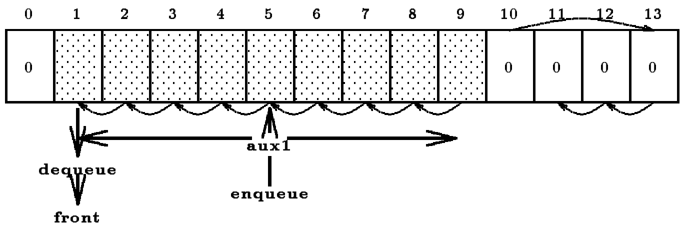

Figure 7: Execution of "Shuffler" Program

3.6.2 Problem Specific Primitives

To demonstrate that it is possible to evolve the desired circular queue a series of runs were made which included "modulus increment" (MInc$n$) terminals. MInc$n$ perform the actions required to implement a circular data structure. Specifically, add one to aux$n$, reduce modulo max, store the answer back into aux$n$ (and return it). In addition "good software engineering practice" was enforced by restricting which primitives could be used by which operation (see Table 2 for details).

In one set of runs, of the 11 that completed, five produced solutions which passed all the fitness tests and correctly implement circular queues. I.e. with powerful primitives the queue problem can be readily solved by GP.

Estimating the probability of a successful run $P(10^4, 42)$ at 5/11 (i.e. 5 successes in 11), the number of runs required to be assured (to within probability 1%) of obtaining at least one solution is 8. This requires $8 \times 10,000 \times 42 = 3,360,000$ individuals to be processed.

3.7 RESULTS WITHOUT MINC

3.7.1 Pass by Reference

In order to allow a modulus increment primitive (cf. MInc$n$) subroutine to evolve, adf1 was changed so that it changes the argument it is passed. E.g. if adf1 implements MInc and aux2 has the value 8, adf1(aux2)

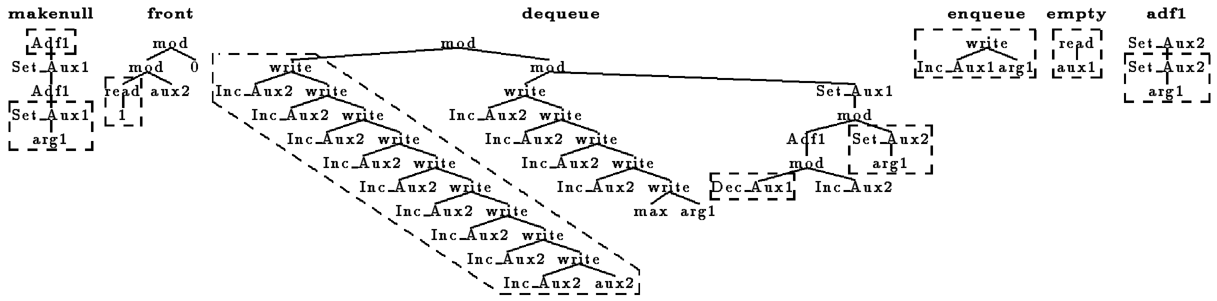

Figure 8: "Shuffler" Program

Table 2: Primitives Used by Each Operation & GP Parameters

Arithmetic	Read only	Update	Initialise
+, −, 0, 1, max, mod, PROG2, QROG2	aux1, aux2, aux3, read	Inc_Aux1, MInc1, Inc_Aux2, MInc2, Inc_Aux3, MInc3, Dec_Aux3, write	Set_Aux1, Set_Aux2, Set_Aux3
Used by all trees	all but adf1	makenull, dequeue, enqueue	makenull
Others Adf1 by all trees but itself	Arg1 by enqueue and adf1	Front by Dequeue	
Panmixia Pop = 10,000, G = 50, Pareto, memory penalty >15, 50% of test data within ±15.7, No aborts			

would set aux2 to 9. In traditional programming languages this is done by passing to the subroutine a reference (pointer) to its argument. In our example, adf1 would be passed a reference to aux2.

In the following experiments pass by reference is implemented by making adf1 set the variable (whose reference has been passed to it) to the value adf1 has calculated. The ADF adf1 continues to return the value as before.

3.7.2 Pareto Fitness Comparison

There was some evidence that the fitness function favored evolution of one operation (empty) above the others. Various ways of weighting the fitness where considered, but Pareto optimality [Gol89, page 197] offered a way of comparing programs without introducing an arbitrary means of combining all their operations into a single fitness. Therefore it was decided to use Pareto fitness rather than explore increasingly complex fitness scoring schemes.

Six criterion were used: the number of tests passed by each of the five operations and the number of memory cells used (above 12). Pareto optimality was combined with tournament selection by using multiple criterion to select the best individual from the tournament group. Where the group contains two or more individuals which dominate the rest of the group (are best on all criterion) but not each other, one of them is chosen at random to be the winner.

Tournaments are still used to decide which individuals are removed from the (steady state) population. However there can now be multiple individuals with different scores which are the best or *elite* (on different criterion) and so elite individuals may be lost from the population as a result of a tournament with other elite individuals. I.e. the population is not *elitist*.

3.7.3 Good Software Engineering Practice

Various measures to encourage evolving genetic programs to follow "good software engineering practice" such as penalizing excessive memory usage and restricting which primitives are used where, have been sketched (see also Table 3). In the final experiment, the adf1 was forced to be "sensible". In particular, it could not yield a constant and it had to transform its input so that its output would not be equal to its input.

These rules are enforced by testing the adf1 part of each program independently of the rest of the program. The adf1 is rejected if any value returned by adf1 is the same as its input or all the answers returned by adf1 are the same. adf1 is tested with the values 0, 1, ... 9 and each answer given by adf1 with these values. I.e. if adf1(9) = 10, then adf1 will be also be tested with a value of 10.

3.7.4 Demic Populations

In the hope of reducing premature convergence, the whole population was treated as a 100×100 square torodial grid. Each grid point contains a single individual and is the center of a 3×3 square *deme* [Col92]. When a new individual is created, its parent(s) are selected from the same deme as the individual it replaces. Tournament selection is used, as before, however the four candidates are chosen (at random with reselection) from the same deme rather than from the whole population.

Table 3: Low Level Primitives Used by Each Operation & GP Parameters

Arithmetic	**Read only**	**Update**	**Initialise**
+, −, 0, 1, max, mod, PROG2, QROG2	aux1, aux2, read	write	Set_Aux1, Set_Aux2
Used by all trees	all but adf1	makenull, dequeue, enqueue	makenull
Others Adf1 by dequeue and enqueue	Arg1 by enqueue and adf1	Front by Dequeue	
Pop = 10,000, G = 100, deme = 3 × 3, Pareto, Memory penalty >12, 50% test data within ±5.0, No aborts			

The failure of runs without demes and without MInc$n$ primitives to evolve circular queues suggests that small demes are required. However only a limited number of fitness functions and parameters have been tried.

3.7.5 Solutions Produced

In one set of 57 runs (using the primitives given in Table 3), six produced solutions which passed all 320 tests. All six solutions use adf1 to implement some kind of MInc. The adf1s evolved are complex; three of them reduce modulo 11 rather than 10.

Subsequent analysis shows that three of the solutions are entirely general solutions to the queue problem. I.e. will pass any legal test sequence. Further, given suitable redefinition of max and sufficient memory, all three could implement an integer queue of any reasonable length (after they were evolved, each passed 32,000 tests with queues of up to 757 items in length). Figure 9 shows how one of the correct programs implements a circular queue of up to nine integers, the code is show in Figure 10.

Analyzing the other three programs shows that whilst they pass all 320 tests, they are not general, i.e. test sequences could be devised which they would fail. This may be a result of reducing the range of values enqueued by too much (50% lie in -5…5).

From the data plotted in figure 11 we estimate $P(10^4, 100)$ at 3/57 (i.e. 3 good solutions in 57 runs). As before we calculate the number of independent runs required to be assured (to within 1%) of obtaining at least one good solution from $P(10^4, 100)$ which yields 86. 86 runs would require up to $100 \times 10,000 \times 86 = 86,000,000$ trial programs to be tested.

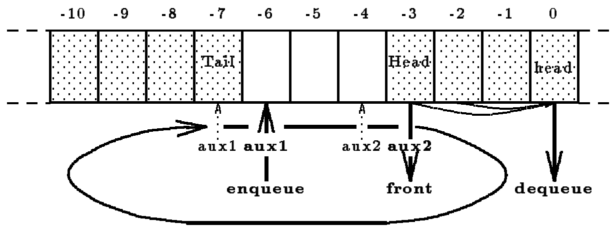

Figure 9: Execution of Evolved Queue Program (2)

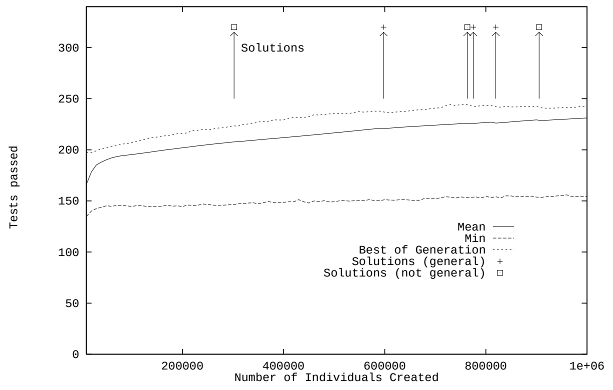

Figure 11: Total tests passed, Means of 57 Queue runs

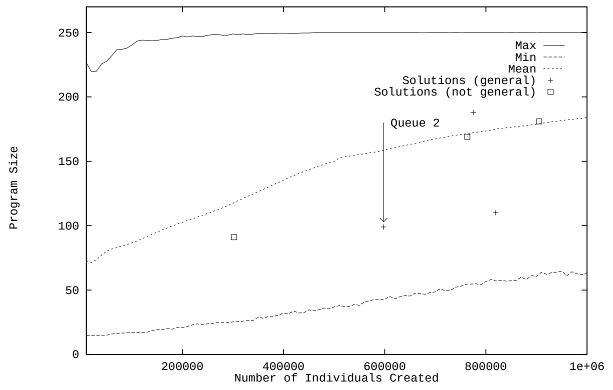

Figure 12: Program Size, Means of 57 Queue runs

4 CONCLUSIONS

The experiments reported herein show that genetic programming, plus indexed (i.e. directly addressable) memory, can evolve programs which implement simple abstract data structures, namely a stack and a queue. Each data structure was implemented by five co-operating but independent procedures. To simultaneously evolve all the procedures, each is represented as an independent tree within the same chromosome.

As anticipated, the stack proved to be easier to evolve than the queue when each had access to problem specific primitives, i.e. the appropriate increment and

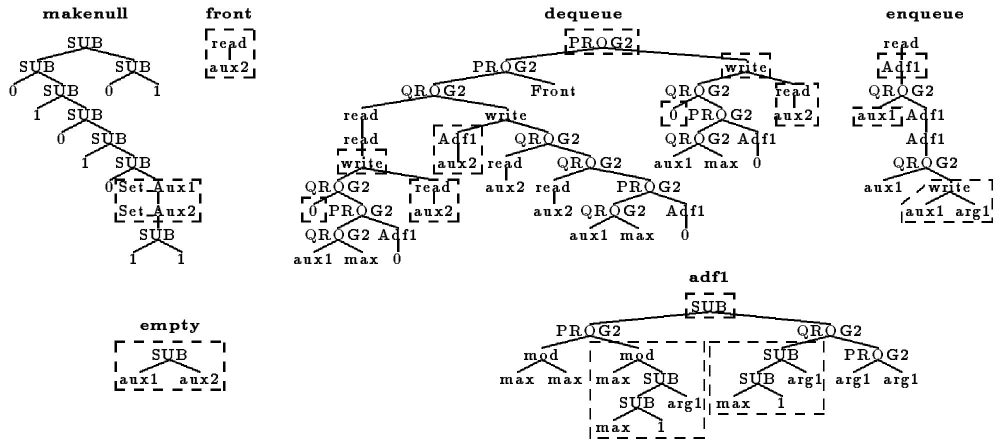

Figure 10: Evolved Queue Program (2)

decrement operations. However such primitives need not be essential. GP still evolved a circular queue (n.b. the more difficult problem) even without the problem specific primitives (take Modulus and Increment, MInc). It was able to do this by evolving them using an evolvable subroutine (an ADF) which used "pass by reference" to update its argument. Not surprisingly, this required considerably more effort than when the primitives were given.

Pareto optimality is a natural way to judge fitness when evolving multiple procedures simultaneously and can be readily incorporated into GP using tournament selection. However further work is required to determine the best way to use it within GP.

"Good software engineering practice" measures where used to encourage the evolutionary process. In particular we restricted which primitives could be used in which tree, penalized excessive memory usage and forcing the ADF to be "sensible".

In these experiments the GP showed a marked tendency to converge to non-optimal solutions. Thus these problems would appear to be "GP deceptive". Partitioning the populations, using demes, was beneficial in this case.

Having solved the stack and queue problems, we intend to study more complex data types, such as lists. Real world problems are more readily solved using abstract data types; we intend to investigate, using a real world scheduling problem, how evolving abstract data types within GP extends the range of problems it can solve.

Acknowledgments

W. B. Langdon is funded by the EPSRC and National Grid Plc. I would like to thank my supervisors (M. Levene and P. C. Treleaven), Tom Westerdale, and Mauro Manela for their critisims and ideas; Andy Singleton for GP-QUICK; and Adam Fraser for GPC++.

References

[AHU87] A V Aho, J E Hopcroft, and J D Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1987.

[Col92] Robert J. Collins. *Studies in Artificial Evolution.* PhD thesis, Artificial Life Laboratory, Department of Computer Science, UCLA, 1992.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison Wesley, 1989.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT press, 1992.

[Koz94] John R. Koza. *Genetic Programming II Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[Sin94] Andy Singleton. Genetic Programming with C++. *BYTE*, February 1994.

[Tel94a] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9. MIT Press, 1994.

[Tel94b] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. *IEEE World Congress on Computational Intelligence*, 1994.