20 Data Structures and Genetic Programming

William B. Langdon

In real world applications, software engineers recognise the use of memory must be organised via data structures and that software using the data must be independant of the data structures' implementation details. They achieve this by using abstract data structures, such as records, files and buffers. We demonstrate that genetic programming can automatically implement simple abstract data structures, considering in detail the task of evolving a list. We show general and reasonably efficient implementations can be automatically generated from simple primitives. A model for maintaining evolved code is demonstrated using the list problem.

20.1 Introduction

Much published work on genetic programming (GP) evolves functions without "side-effects" to learn patterns in test data. In contrast human written programs often make extensive and explicit use of memory. Indeed memory in some form is required for a programming system to be *Turing Complete*, i.e. for it to be possible to write any (computable) program in that system. However inclusion of memory can make the interactions between parts of programs much more complex and so make it harder to produce programs. Despite this it has been shown GP can automatically create programs which explicitly use memory [Teller 1994].

In both normal and genetic programming considerable benefits have been found in adopting a "structured approach". For example [Koza 1994] shows the introduction of evolvable code modules (automatically defined functions, ADFs) can greatly help GP to reach a solution. We suggest that a corresponding structured approach to use of data will similarly have significant advantage to GP.

Earlier work has demonstrated that genetic programming can automatically generate simple abstract data structures, namely stacks and queues [Langdon 1995a]. That is, GP can evolve programs that organise memory (accessed via simple `read` and `write` primitives) into data structures which can be used by external software without it needing to know how they are implemented.

This chapter shows it is possible to evolve a list data structure from basic primitives. [Aho, Hopcroft and Ullman 1987] suggest three different ways to implement a list but these experiments show GP can evolve its own implementation. This requires all the list components to agree on one implementation as they co-evolve together.

Section 20.3 describes the GP architecture, including use of Pareto multiple component fitness scoring (20.3.4) and measures aimed at speeding the GP search (20.3.5). The evolved solutions are described in Section 20.4. Section 20.5 presents a candidate model for maintaining evolved software. This is followed by a discussion of what we have learned (20.6) and conclusions that can be drawn (20.7).

Appendix 20.A describes two techniques for reducing run time.

Table 20.1
Definitions of the Ten List Operations

Makenull	Make the list an empty list and return position given by End.
Retrieve(p)	Return the element at position p.
Insert(x, p)	Insert x at position p, moving elements at p and following positions to the next higher position.
Delete(p)	Delete the element at position p, moving elements at $p+1$ and following positions to the previous lower position.
End	Return the position following the last element of the list.
First	Return the position of the first position. If the list is empty, return the position given by End.
Next(p)	Return the position following position p.
Previous(p)	Return the position before position p.
Locate(x)	Return the position of the first element containing value x. If x is not in the list at all then return the position given by End.
Printlist	Print the elements in their order of occurrence.

20.2 The List Problem

An integer list data structure [Aho et al. 1987] has ten component operations which are summarised in Table 20.1. Note that a list is a generalization of a stack and a queue, i.e. both can be implemented using a list by restricting access to its ends.

The immediate goal is to evolve a list which works correctly with a limited number of elements, nine was chosen as sufficient to demonstrate the principle. But as we shall see (in Section 20.4) GP is capable of evolving lists of any size.

20.3 Using GP to Solve the List Problem

20.3.1 Architecture

The architecture was chosen to encourage the ten list operations to co-evolve, each solving their own part of the problem and together forming a complete solution. It does this by binding them together to form individuals in the GP population. Thus each individual contains code for all ten operations and is a trial solution for the whole list.

As each operation has different requirements the GP crossover operator was modified to ensure it only moves subtrees between code for the same operation. Thus code can evolve to specialise to just one requirement (i.e. perform one operation). This is implemented by representing each operation with a separate tree within the individual. This representation can easily be extended to allow Automatically Defined Functions (ADFs) [Koza 1994] by allocating each ADF an additional tree.

20.3.2 Automatically Defined Functions

Work on the queue has shown that co-evolving code which has some functionality in common can be eased if there is a shared ADF which can evolve to provide common code for the

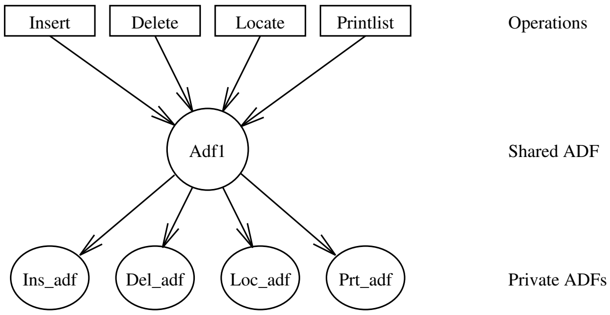

Figure 20.1
ADF Calling Hierarchy Available to Solve the List Problem

common functionality. This appears to be far easier than expecting the same functionality to evolve twice in separate locations.

One common requirement is for an addressing scheme for the list elements, however End, First, Next and Previous already collectively provide addressing actions. To take advantage of this, other parts of the list are allowed to call them, i.e. their code both implements the operation and is an ADF.

Work on the queue indicates that it helps, if functions which calculate the next value of their argument use a pass by reference calling mechanism. This avoids evolving code to write the return value into the argument after every call. Therefore operation/ADFs that have arguments (i.e. Next and Previous) update them directly.

Four operations (Insert, Delete, Locate and Printlist) all process the list element by element. A single ADF (Figure 20.1) was provided in the hope that it would evolve to meet this common need. As each operation processes list elements differently the ADF is parameterised, using a private ADF for each operation. When the main ADF is called, it is passed a reference to the corresponding private ADF, which it in turn may call (using the reference). To avoid additional arguments each private ADF may, where necessary, have access to the arguments of its operation as well as its own (effectively it is in the scope of its operation).

20.3.3 Terminals and Functions

The terminals and functions were chosen to make the task of evolving a list as easy as possible and follow on from those used when solving the queue problem. Tables 20.2 and 20.3 show where they may be used and describe what they do.

Thirty one memory cells are provided, which should suffice for lists of up to nine elements. These are numbered $-15 \ldots 15$, which allows evolved code to use negative as well as positive addresses. Code may also use the auxiliary variable (`aux1`), possibly to store the length of the list.

Table 20.2
Terminals and Functions Available to Solve the List Problem

Tree	Terminals and Functions Available
Makenull	`PROG2, write, Set_Aux1, End`
Retrieve	`arg1, read`
Insert	`PROG2, aux1, adf1, Next, ARG1, ARG2, write`
Delete	`PROG2, aux1, adf1, Next, ARG1, Prev`
End	`aux1`
First	`aux1`
Next	`arg1`
Previous	`arg1`
Locate	`adf1, First, ARG1`
Printlist	`adf1, First`
Adf1	`arg1, aux1, forwhile, i0, FUNC, End`
Ins_adf	`arg1, swap`
Del_adf	`arg1, swap, ARG1, Next`
Loc_adf	`arg1, ARG1, read`
Prt_adf	`arg1, read, print`
All trees may contain +, −, 0, 1 and `max`	

The requirement to process multiple list elements means the GP must support iteration (loops) or recursion, either implicitly (e.g. using block memory move functions) or explicitly. It was decided not to use high level functions but instead to require the GP to explicitly evolve iterative structures (loops appeared simpler to implement than recursion). Fears that the loop primitive, `forwhile`, could cause excessive run time lead to: forbidding nested loops; loops only being allowed where they appear to be required (i.e. the ADF); and a limit (32) on the number of iterations. The limit was set as low as possible but still allows loops to span all the available memory.

The following syntax rules are imposed with the aim of aiding the evolutionary process by forbidding exploration of apparently infeasible programs without the cost of evaluating their fitness and also ensuring essential primitives are not lost from the population:

- Prt_adf must contain at least one `print` function.

- The loop index, `i0`, may only appear inside the third argument of `forwhile`, i.e. the loop. Nested loops are prevented by a syntax check.

- Adf1 must contain at least one loop, which must contain at least one `i0`.

- It would appear to be sensible for code to use its arguments. Where feasible, syntax checks are used to encourage this. Thus Retrieve, Next, Previous, Adf1, Ins_adf, Del_adf, Loc_adf and Prt_adf must contain at least one `arg1` terminal and Adf1 must contain at least one `FUNC` (Table 20.3) primitive.

Table 20.3
Actions Performed by Terminals and Functions

Primitive	Purpose
`max`	constant 10 ($\geq$ max list size).
`PROG2`(t,u)	evaluate t; **return** u
`arg1`	argument of current operation or ADF, but:
`ARG1, ARG2`	arguments of Insert, Delete, Locate or Printlist.
`aux1`	an auxiliary variable (i.e. in addition to indexed memory).
`Set_Aux1`(x)	`aux1` = x; **return** `aux1`

`forwhile`(s,e,l)	**for** `i0` = s; `i0` $\leq$ e; `i0`++
	    if timeout **exit loop**
	    if l returns zero **exit loop**
	return `i0`

`FUNC`	call private ADF of operation which called Adf1.
`print`(d)	**if** space in print buffer (10) copy d into it; **return** number of items in it **else** evaluate d; **return** 0
Indexed memory is held in store[$-l \ldots +l$], where l = 15, i.e. a total of 31 cells.	
`read`(x)	**if** $\lvert x\rvert \leq l$ **return** store[x] **else return** 0
`write`(x,d)	**if** $\lvert x\rvert \leq l$ store[x] = d; **return** original contents of store[x] **else** evaluate d; **return** 0
`swap`(x,y)	**if** $\lvert x\rvert \leq l$ **and** $\lvert y\rvert \leq l$ exchange contents of store[x] and store[y] **if** $\lvert x\rvert > l$ **and** $\lvert y\rvert \leq l$ store[y] = 0 **if** $\lvert x\rvert \leq l$ **and** $\lvert y\rvert > l$ store[x] = 0 **return** 1

20.3.4 Pareto Optimality

Most GPs (and indeed genetic algorithms in general and other search techniques) use a scalar fitness function where each individual is given a single measure of its usefulness. An alternative is to use a multi-dimensional fitness measure where each fitness dimension refers to a different aspect of the trial solution. In our case the success of each operation is a different dimension within the trial program's fitness.

Pareto optimality provides a means to compare multiple objectives without prioritising the objectives [Goldberg 1989]. Fitness values are compared component by component. If one is no worse than the other in every case and better in at least one case then it is said to dominate the other. For example in Figure 20.2, point 2 dominates B but does not dominate A.

Pareto scoring means individuals which make an improvement on any part of the problem tend to be preferred, whereas a scalar fitness will tend to require each improvement to match or exceed any deterioration in all other parts of the problem. Whether an improvement is

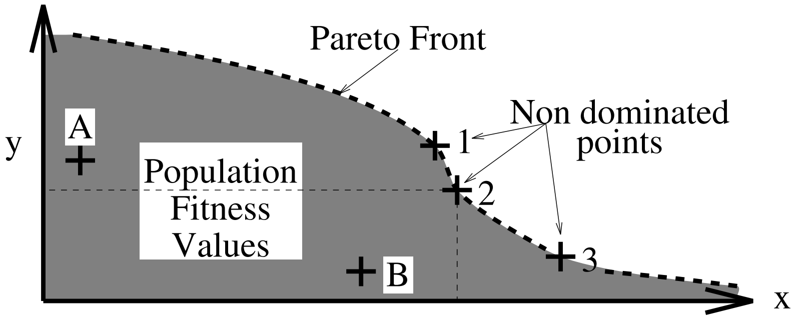

Figure 20.2
Two Dimensional Example of Pareto Optimality and the Pareto Front

more important than a deterioration is given by scaling parameters within the fitness function. Consequently setting them is complex and must be done with care. To some extent Pareto fitness avoids this problem.

With Pareto scoring, a single population can contain several hundred different fitness values (or niches), each of which is the best on some combination of criteria [Horn, Nafpliotis and Goldberg 1993]. This encourages crossover between individuals that are good at different parts of the problem, which may produce offspring that are relatively fit on more of the problem.

20.3.5 Directed Crossover

As Figures 20.3 and 20.4 show, GP solves the different parts of the list problem at different times. If it were known that a fragment of code was working well it would seem wasteful to perform crossover in it. Of course GP does not know for certain if code is correct, however the crossover location can still be guided by the program's current behaviour (as this may be misleading 10% remain unguided). A number of papers show (albeit on very different problems) benefits in using either current behaviour (Chapter 9), program syntax (Chapter 19 and [D'haeseleer 1994]) or evolving program syntax [Whigham 1995] to bias crossover or other genetic operators. Our mechanism succeeds in dynamically redistributing crossover locations to code in need of improvement as the population evolves. At present it only considers code at the level of individual operations or ADFs but could be refined.

The mechanism uses knowledge of the first parents fitness and the number of times the trees in it were executed and whether they appeared to be successful or not to probabilistically bias the choice of which tree the crossover occurs in. It avoids trees which are believed to be correct (see Section 20.3.6.3), that have never been executed or those that passed all their fitness tests. Otherwise it is biased to choose trees that appear to fail most often [Langdon 1995b].

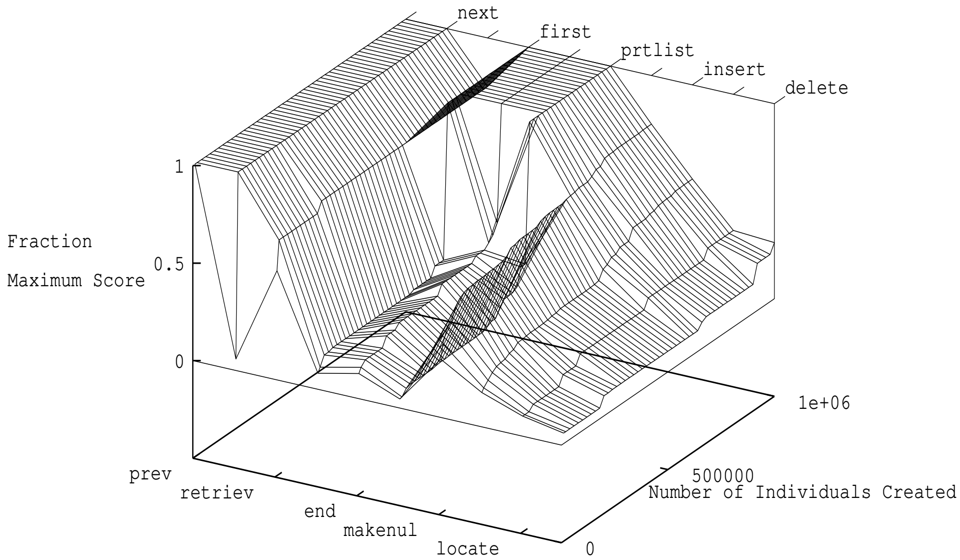

Figure 20.3
Evolution of the Maximum Score in the Population for Each List Operation, Typical Run

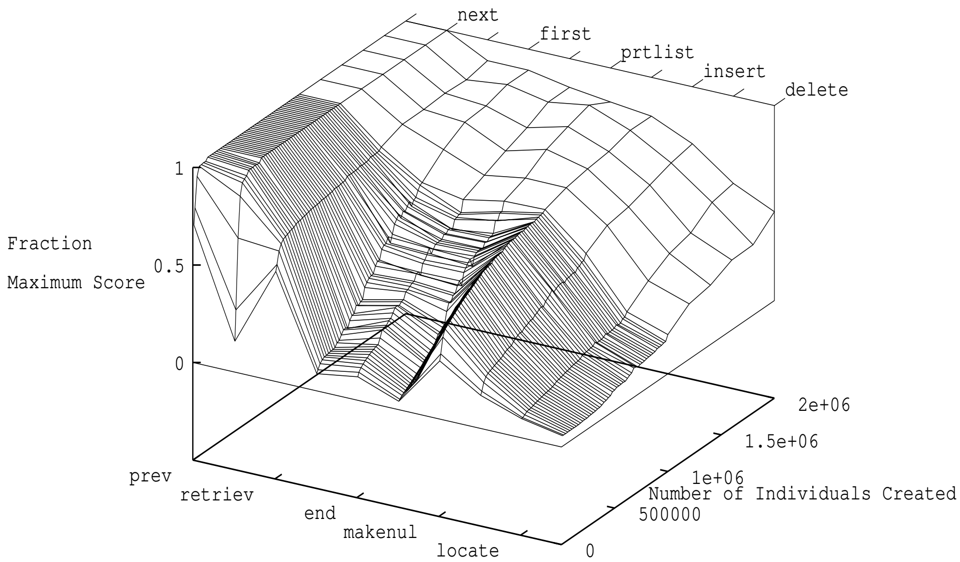

Figure 20.4
Evolution of the Maximum Score in the Population for Each List Operation, Means of 56 Runs

20.3.6 Fitness Function

The fitness of each individual is determined by running it on 21 fixed test sequences containing in total 538 operations. Tests are grouped into (a total of 167) subsequences which call several operations and cross check the values returned by them. If the checks are passed we increase the program's fitness along each dimension corresponding to an operation called in the subsequence.

No preset design or pattern of memory usage is imposed by the fitness tests. This undoubtedly makes the problem more difficult but we wish to show an implementation can be automatically generated. This flexibility makes testing more complex as often it is impossible to say the answer returned by an operation is correct, until it can be compared with answers returned by other operations. The tests only cover correct operation of the list. Error trapping, such as detecting the deletion of non-existing list elements is not covered as yet.

All storage, i.e. the indexed memory and `aux1`, is initialized at the start of each test sequence (i.e. 21 times). Not only does this ensure individuals behaves predictably and prevent "leakage" of information between members of the population, it also gives imperfect programs multiple opportunities to pass some of the tests from a clean state. If this was not done, the program would start each test sequence with `aux1` and memory values disrupted by previous attempts to run it.

In half the test sequences the indexed memory is initialized to zero (`aux1` is always set to zero). In the other half, the indexed memory is initialized to a random but fixed data pattern, which is different for each test sequence. This is to discourage the GP from evolving partial solutions which exploit the value zero to indicate memory cells are unlikely to have been written to and so are empty. In earlier experiments such programs had been found, they appeared to work until they inserted zero into the data structure. The random data patterns have the same distribution of values as that inserted into the list.

20.3.6.1 Test Data Values

Values to be entered into the list are specified in the test sequences. With the stack and queue it was advantageous to select data values at random from a large range of both positive and negative values but also to ensure that that there were a reasonable number of small values, particularly zero and one. This can be achieved by generating random numbers uniformly in the range $0 \ldots \pi$ and then taking their tangents. This produces equal numbers of positive and negative numbers, some very large but half lie in the range $-1.0 \ldots +1.0$. Before they are converted to integers they are multiplied by a factor F which reduces the number of times zero occurs.

F was set to 15. This ensures half of data values inserted in the list lie in the range of legal memory indexes (i.e. ± 15).

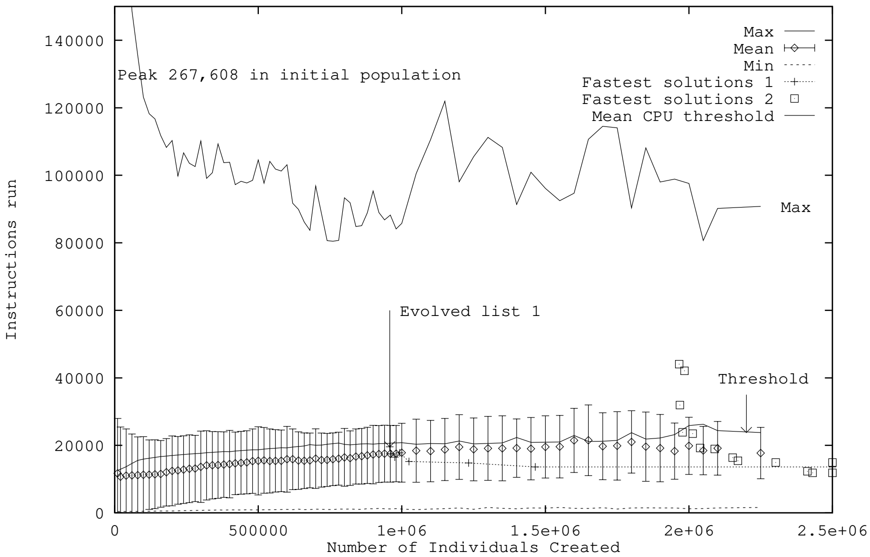

Figure 20.5
Evolution of the number of primitives executed during fitness testing on the List problem, means of 56 runs. Error bars indicate one standard deviation either side of the population mean. The fastest solutions evolved during the two successful runs and the population mean of the threshold above which the CPU penalty is applied are also shown. The minimum number of instructions are executed by relatively unsuccessful programs as these are run on few tests.

20.3.6.2 CPU and Memory Penalties

CPU Penalty

The number of instructions (i.e. function calls and terminals evaluated) are counted as each individual is tested. The mean number per operation used is calculated and becomes a penalty component in the Pareto fitness. However individuals which use less than 120 instructions per operation have zero penalty.

This scheme was introduced as it was anticipated that the `forwhile` loop could lead to very time consuming fitness evaluations and so excessively long run times. As Figure 20.5 shows it has had the anticipated benefit of curbing program run time and also causes the evolution of near parsimonious code (Figure 20.8). This is in dramatic contrast to the queue (which had no CPU or space penalties) where programs rapidly grew to the limit of the available space. However the threshold has to be chosen with care to avoid over penalizing constructs (like loops) which have a high CPU cost but appear not to help achieve higher fitness levels until later in the evolutionary process. Therefore the threshold was deliberately set high at $8 \times$ fastest program that might evolve.

Table 20.4
Key Parameters used to Solve the List Problem. Other parameters are as [Koza 1994, page 655]

Fitness Cases	538 trees run in 21 sequences. 167 consistency tests.
Fitness Scaling	Each tree scored independently using Pareto comparison, Memory usage above minimum (12 cells) and CPU usage above 120 per test run are Pareto fitness penalties.
Parameters	Steady state population of 10,000. Max gens 100. Max prog size 500. Max initial tree size 50. Elitist Pareto Tournament group 4. Niche population sample size 81. 90% directed crossover. 31 memory cells (addressed -15 … +15). 50% data values within ± 15.

Memory Penalty
When evolving a queue, memory hungry partial solutions evolved. Therefore an excessive memory usage penalty was introduced to dissuade the population from evolving down this blind alley. A threshold of using 12 cells must be passed before this penalty applies (12 being more than sufficient for a list of nine items). This penalty was retained for the list and such memory hungry behaviour has not been observed. This might mean the penalty is working or such solutions are not common in which case the penalty may be unnecessary or too restrictive.

In order to search for faster solutions or ones that use less memory, after the first solution has been found in a run, both penalties are increased by setting their thresholds to zero. The new fitness function is applied to members of the existing population as well as newly created individuals.

20.3.6.3 Sufficient Testing
Certain points in the fitness tests are marked with an operation's name. If they are reached without any errors occurring this is taken to mean testing of that operation is complete and it is given its maximum possible score (i.e. the score it would have if every consistency check was passed), regardless of any subsequent errors (which are assumed to be the fault of one of the other operations). This is applied to Retrieve, End, First, Next, Previous and Printlist.

20.4 Results

In a group of 56 runs, two produced solutions which passed all the tests. All runs completed at least 100 generation equivalents with the promising ones being continued (the longest run being 300 generations).

Like the stack and the queue, solutions have been found which not only pass all the tests, but subsequent analysis shows to be correct and general, i.e. given sufficient memory would correctly implement a list of any finite size. Figure 20.6 shows the first of these.

Figure 20.7 contains simplified pseudo code for four of the operations and shows GP has implemented the list as an array, with Insert opening a gap for the new data element by moving part of the list up one memory element. Delete moves the top part of the list down one element overwriting the element to be deleted.

Solutions were also found which exploit the finite size of the test set, in that they are able to pass all of the tests but are not entirely general. Interestingly the same GP run found solutions which are general less than five generations later. Each program contains redundant code, i.e. code that can be removed yielding a shorter program but with the same functionality. In Figure 20.6 the essential code is highlighted by boxes.

On continuing the evolutionary process, shorter solutions with reduced CPU cost were found, see Figures 20.5 and 20.8. In the first successful run reductions of 20% in program length and 30% in the number of instructions required to complete the test sequences were produced. In the second case the changes were more dramatic with program length falling by two thirds and number of instructions to about a quarter of the first solution found. The shortest solutions reported by both runs have similar lengths and CPU.

The evolution of solutions which use fewer instructions is exactly what the increase in the CPU penalty (described in Section 20.3.6.2) was designed to achieve. However if the penalty was not increased there was some reduction due to random fluctuations. The increase in penalty produced bigger effects in these runs but it is not known if this is generally true.

(The data after generation 100 are more sparse. To avoid excessive variability in Figures 20.4, 20.5 and 20.8 only data referring to fifteen or more runs are plotted).

Figure 20.9 shows the number of various primitives in a typical population as it evolves relative to their abundance in the initial random population. It shows a number of primitives become very rare and indeed nine are lost entirely from the population. In 27 runs out of 56 after 100 generations less than 10% of the population contained a primitive required in the solutions that were found. Of these 27, 16 runs lost all of one or more such primitives. When less than 10% of the population contain a primitive its numbers may recover, however often its frequency continues to fall and it becomes extinct.

From Figure 20.8 a crude estimate of the probability of a solution being found by generation 200 when using a population of size 10,000, can be estimated at 2/56. From this the number of GP runs required to be assured (with 99% probability) of obtaining at least one solution can be estimated to be 127 [Koza 1992, page 194]. This would require a total of up to $2\ 10^6 \times 127 = 2.54\ 10^8$ trial programs. N.b. this must be treated as an estimate, not an exact figure.

20.5 Software Maintenance

Software maintenance is the problem of ensuring existing code continues to operate effectively. This may involve correcting bugs in existing code or adapting code to new requirements. Automatically generated code (such as produced by a high level language compiler) may be difficult to maintain at the code level and it is common practice to change the inputs

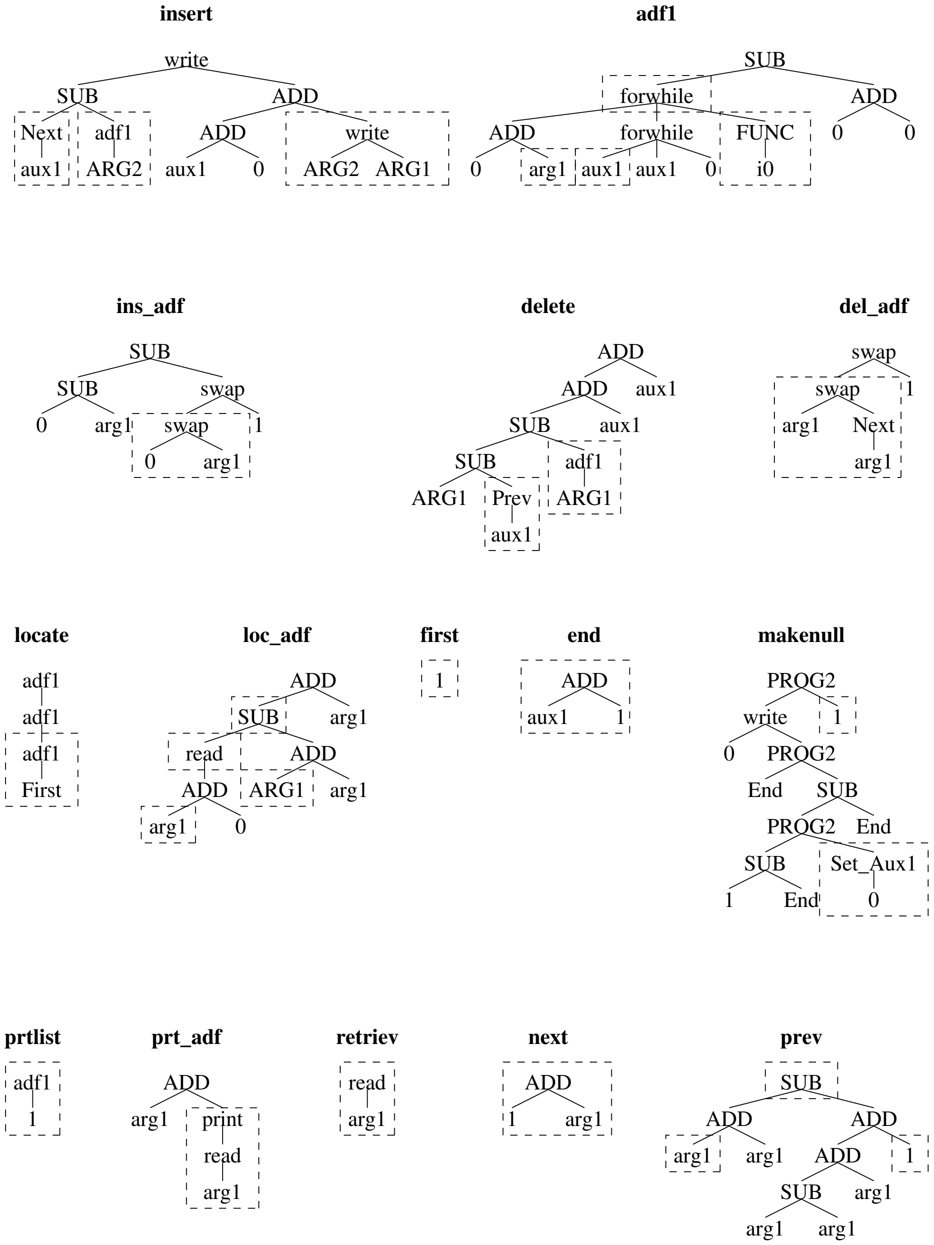

Figure 20.6
First Evolved Solution to List Problem

```
Insert
aux1++
for i0=ARG2; i0≤ aux1; i0++
      swap 0 i0
store[ARG2] = ARG1
```

```
Printlist
for i0=1; i0≤ aux1; i0++
      print store[i0]
```

```
Delete
aux1−−
for i0=ARG1; i0≤ aux1; i0++
      swap i0 (i0 + 1)
```

```
Locate
for i0=1; i0≤ aux1; i0++
      if store[i0] = ARG1 return i0
return aux1 + 1
```

Figure 20.7
Simplified Pseudo Code Equivalent to First Evolved Solution to List Problem

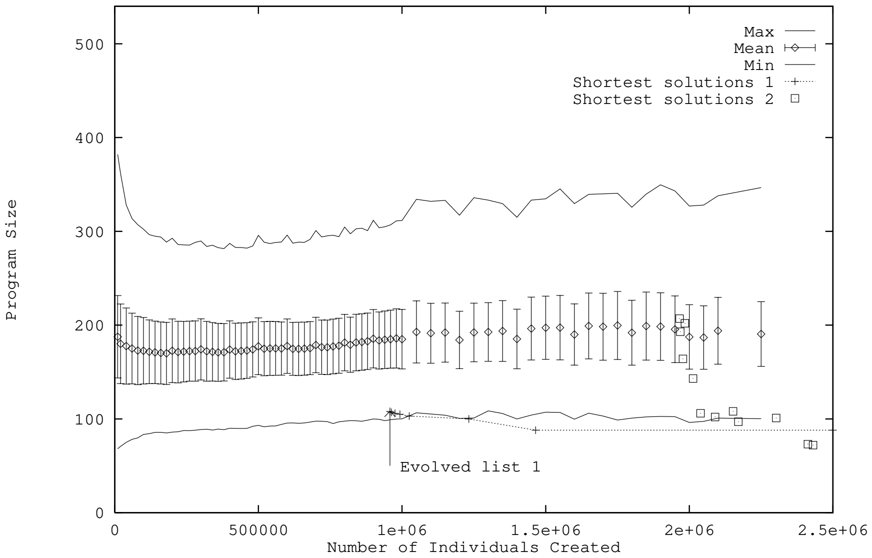

Figure 20.8
Evolution of program size, means of 56 runs. Error bars indicate one standard deviation either side of the population mean. The shortest solutions evolved during the two successful runs are also shown.

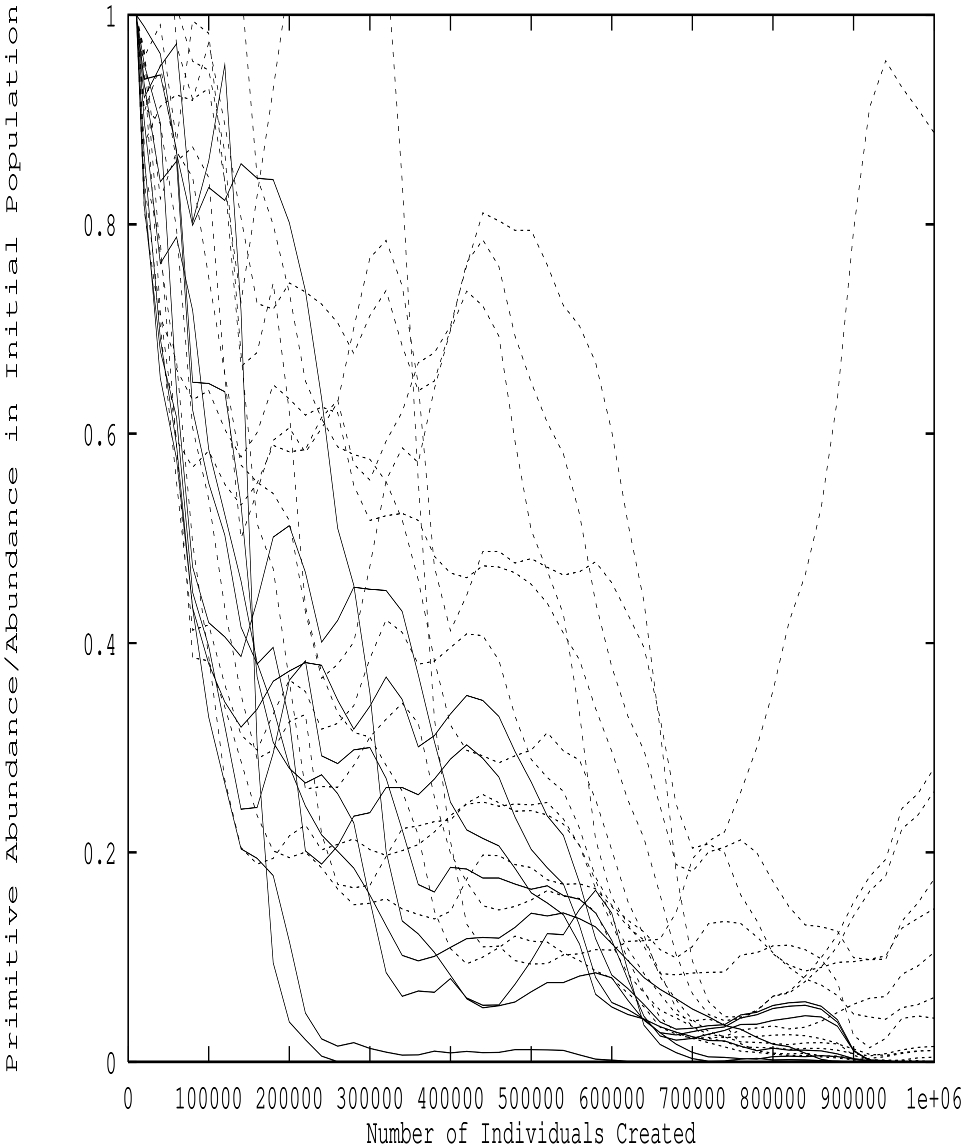

Figure 20.9
Evolution of the frequency of rare primitives (i.e. terminals and functions where the number of them in the population falls below 10% of its original value). These graphs plot the ratio of the number of them to the no. of them in the initial population for a typical run. The nine primitives which are lost completely from the population are shown as solid lines, the others are shown as dotted lines.

to the code generator (i.e. the source code) and run it again. It is anticipated that evolved code will also be difficult to maintain, so maintenance may be performed by changing the inputs to the automatic code generator (i.e. the GP) and running it again.

This section uses the list problem to demonstrate the following model for maintaining evolved code:

1. Start with the original fitness function and the population that contained the solution to the original problem (this should avoid solving the new, possibly harder, problem from scratch),

2. Write additional fitness tests for the new functionality,

3. Expand the existing individuals within the population to include random code for the new functionality,

4. Evolve the expanded population with both the original and new fitness tests.

To use the list problem as a test bed for this model, it is split into two, one part representing the original problem and the other the new requirements. The first is all the operations except Locate and Delete, which represent the new requirements. Locate and Delete were chosen in the expectation that once Insert and Printlist operations are working, being similar, Locate and Delete could be readily added. As they are also the most difficult operations (see Figures 20.3 and 20.4), removing them from the first phase should have the advantage of speeding it up allowing more time to be spent on the maintenance phase.

Each GP run starts as before (Section 20.3) except in the first phase the new fitness function does not test either Delete or Locate (it comprises 14 test sequences and a total of 426 operation calls and 118 consistency checks). After a solution to the smaller problem has been found, solutions are allowed to spread through the population by continuing the run until at least 1,000 other individuals which solve the smaller problem (i.e. pass all 118 checks) have been found before proceeding to the second phase.

At the start of the second phase the trees for Delete and Locate (and their associated ADFs) in every individual in the population are re-created at random. The other eleven trees in each individual are not changed. So we start the second phase with every individual being a hybrid of code that is adapted to the smaller problem (but need not be an exact solution to it) and random code. In the second phase, the first 14 test sequences are augmented by 7 more designed to test Delete and Locate, they contain 113 operation calls and 41 consistency checks. The population is allowed to evolve as before. The directed crossover mechanism (Section 20.3.5) ensures crossovers are allowed in every tree but are weighted towards the newly introduced random code.

20.5.1 Results

In a group of 59 runs, five produced solutions which passed the first set of tests. In these five, evolution was allowed to continue for between 40 and 88 generations, during which two runs found programs which pass the second phase of testing. Both runs produced general programs which implement a list and have a similar structure to those produced in the first experiment. As with the first experiment, on continuing the evolutionary process both runs found shorter solutions and solutions which took fewer instructions to complete the test cases.

From finding the first solution to the first part of problem to starting the second took between 3.8 and 7.2 generation equivalents and the first solutions to the whole problem appeared 8.2 and 13.4 generations later (i.e. 16.3 and 25.8 after the first solutions to the first part).

The probability of a solution being found by generation 25.8 when using a population of size 10,000, is estimated to be 2/5 (further work is needed to verify this estimate). Using this, the number of individuals which need to be evaluated in order to be 99% sure of at least one solution is $2.58\ 10^6$. Or $1/100^{th}$ of the effort to solve the whole problem.

20.6 Discussion

The key to successful human produced software is using abstraction to control the complexity of each task in hand. I.e. in each task, being able to use objects without considering in detail how they interact. The objects are abstractions of lower level code or data, which are in turn abstractions of still lower levels. Thus a programmer may use a file while knowing nothing about how to program the disk drive on which it is stored. Indeed the program can continue to work even if the file is moved to another disk drive, even if it is of a different type, even a type which had not been designed when the program was written.

Genetic programming, with its undirected random program creation, would appear to be the anathema of highly organised software engineering. It is an evolutionary technique, using only information in its current population. It does not plan ahead or use a global top-down design but we have already seen, via ADFs and other techniques, it too can gain considerable benefit from functional abstraction.

While work to date has concentrated on functional abstraction, we argue that techniques which allow GP to take advantage of data abstraction will be essential to enable it to scale up and tackle large real problems. We have started by showing GP can produce simple structured data types as these seemed achievable with current GP techniques and resources and should serve as a starting point to demonstrate GP using data abstraction, perhaps via a methodology for GP using abstract data types. They also contain many of the problems associated with program production. For example producing multiple co-operating modules and how to test code, such as push, which only acts via side-effects that are not immediately accessible.

There are always practical limits on the amount of testing that can be performed. Therefore we decided to evolve data structures of a fixed modest size (nine items) and this was built into the fitness tests. However in all but one case, programs which pass all the fitness tests and which subsequent analysis shows are correct solutions to the problem of fixed size, are also implementations of the general data structure. I.e. they could implement the data structure of any fixed size (given sufficient memory and suitable definition of `max`). This is very encouraging. Perhaps general algorithms are easier for GP to find?

While genetic programming appears to find the list problem hard, in terms of the number of individuals generated, it is only about three times more difficult than the queue, despite requiring the co-evolution of ten operations rather than five. There are many differences between the GPs used to solve these problems (such as the syntax restrictions, Section 20.3.3) which no doubt play a part in the difference, none the less so small a rise in problem difficulty given the change in size is interesting.

It is tempting to ascribe the GP's difficulty in finding solutions to the many cases where vital primitives are removed by evolution from the population. The size of the population makes it very unlikely that the complete loss of a primitive from the population (Section 20.4) is due to random chance (known as "genetic drift"). However this should be regarded as a symptom of a deeper problem: they are not justifying their presence in the *current* population.

Their loss indicates they have below average fitness, i.e. in many populations, the fitness function is being deceptive and leading the GP towards some local optima and away from correct solutions. Thus simply adding more of them to the population via a mutation operator would not be expected to solve this problem directly. (However mutation might be beneficial by allowing the population to retain them whilst it evolves beyond the deceptive local optima or via other effects such as incorporating an element of hill climbing, see Chapter 2 and [Iba, de Garis and Sato 1994]). Their below average fitness appears to be due to the CPU penalty, which introduces a small selection pressure against "introns" (see Chapter 6) or code of no immediate purpose and this eventually succeeds in removing all of certain primitives.

Measures to retain population diversity such as fitness niches or demes appear to be necessary to give the evolutionary process time to assemble the primitives into high value building blocks from which complete solutions can be assembled so enabling it to escape from local optima. Overlapping demes, which constrain mates to be selected only from near neighbours (similar to those in [Tackett and Carmi 1994]) succeeded in delaying the lost of primitives but spreading the population out across the fitness landscape using a "fitness sharing function" [Horn et al. 1993] was more effective in delaying the losses for longer and so was used.

While fitness niche appear to combat the effects of deceptive fitness functions to some extent, the many non-dominated solutions within the population mean the primary selection pressure is to find relatively un-populated fitness niches, rather than better ones. It also means almost all crossovers occur between disparate individuals. It is unclear how beneficial this is. [Ryan 1994] suggests benefits for disassortive mating, while [Harvey 1992] suggests (for a variable length but linear GA) better results may be obtained by breeding between similar programs. He suggests this will produce smaller improvements at each stage but more progress in the long run.

One of the lessons, for multi-part programs, from the queue was while ADFs are useful, it may be better to ensure the evolutionary pressures on them do not pull in more than one direction by avoiding an ADF being used in two completely different ways. For example in the queue problem the enqueue and empty operations have little in common, when they shared the same ADF it appeared to try and satisfy both and so failed to develop any clear functionality and no overall solutions where found. Accordingly the use of ADFs was constrained (Table 20.2). This up front GP design and the use of syntax restrictions (Section 20.3.3) was intended to help the GP, it would be interesting to see how far the GP would get with fewer restrictions.

20.7 Conclusions

The importance of abstract data structures to software engineering is well recognised. Our experiments show genetic programming, using indexed memory, can automatically implement integer list structures, co-evolving all ten components simultaneously. The list being a generalisation of stack and queue data structures (which have already been evolved) and is more complex than either.

As with earlier work, generic programs have been automatically created that not only solve the problem on which they were trained but which (with sufficient memory) implement the abstract data structure of any size.

A model for maintaining evolved software based on population re-use has been demonstrated and (in one example) considerable savings shown compared to evolving a solution to the new requirement from scratch.

Program execution time can be included as a Pareto component of fitness and led to shorter and more efficient programs. It appears a consequence of the reduced program size was a marked tendency for the number of certain terminals and functions within the population to fall to low values or extinction.

It has been shown that a genetic programming population using Pareto tournament selection, in conjunction with comparison with (a sample of) the rest of the population, can stably support many fitness niches.

Whilst this work has shown fitness niches, CPU penalties and biased choice of crossover points were effective when evolving a list, further work is required to demonstrate to what extent they are generally useful.

20.A Implementation Issues

20.A.1 Reusing Ancestors Fitness Information

Partial scores for each test sequence are saved. If crossover changes a tree which is not executed in a particular test sequence then the new program's score on that sequence must be the same as its first parent's. This is recognised and the partial score of the parent is used so avoiding running the test. The savings produced are dependant upon the test sequences, GP parameters and other time saving techniques (20.A.2), nonetheless run time was halved in some cases.

The technique could be extended to cover other cases where a child's score must be identical to that of one of its parents. For example children produced by crossovers which occur within introns, must behave identically to their parent and so have identical fitness.

20.A.2 Caches

Where the primitives are free from side effects, [Handley 1994] demonstrates considerable performance improvements can result from caching results produced by subtrees. `read`, `write` and other primitives *do* have side effects, never the less caching can be used if parts of the program are known to be free of side effects. Hit ratios of between 70% and 97% where obtained with caches on End (1000 words), First (200), Next (1000) and Previous (200), which reduced runtime by about 50%.

The C++ code used in these experiments is based upon GP-QUICK and is available, for research and educational purposes, via anonymous ftp; node `cs.ucl.ac.uk` directory `genetic/gp-code`.

Acknowledgments

W. B. Langdon is funded by the EPSRC and The National Grid Company plc. I would like to thank M. Levene and P. C. Treleaven for their critisims; Andy Singleton for GP-QUICK; and Lee Spector, Maarten Keijzer, Thomas Haynes, Frederic Gruau and Peter Angeline who all helped me by reviewing this chapter.

Bibliography

Aho, A. V. (1987) J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley.

D'haeseleer, P. (1994) Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261. IEEE Press.

Goldberg, D. E. (1989) *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley.

Handley, S. (1994) On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, 154–159. IEEE Press.

Harvey, I. (1992) Species adaptation genetic algorithms: A basis for a continuing SAGA. In F. J. Varela and P. Bourgine, eds. *Toward a Practice of Autonomous Systems, Proceeding of the first European Conference on Artificial Life (ECAL)*, pages 346–354. MIT Press.

Horn, J (1993) N. Nafpliotis, and D. E. Goldberg, Multiobjective optimization using the niched pareto genetic algorithm. IlliGAL 93005, University of Illinois, 104 South Mathews Avenue, Urbana, IL 61801

Iba, H. (1994) H. de Garis, and T. Sato, Genetic programming with local hill-climbing. In Y. Davidor, H-P. Schwefel and R. Männer, eds. *Parallel Problem Solving from Nature III*, 334–343. Springer-Verlag.

Koza, J. R. (1992) *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press.

Koza, J. R. (1994) *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

Langdon, W. B. (1995a) Evolving data structures using genetic programming. In L. Eshelman, ed. *Genetic Algorithms: Proceedings of the Sixth International Conference*, 295–302. Morgan Kaufmann.

Langdon, W. B. (1995b) Directed crossover within genetic programming. Research Note RN/95/71, University College London, Gower Street, London WC1E 6BT, UK.

Ryan, C. (1994) Pygmies and civil servants. In K. E. Kinnear, Jr., ed. *Advances in Genetic Programming*, chapter 11. MIT Press.

Tackett, W. A. (1994) and A. Carmi, The donut problem: Scalability and generalization in genetic programming. In K. E. Kinnear, Jr., ed. *Advances in Genetic Programming*, chapter 7. MIT Press.

Teller, A. (1994) The evolution of mental models. In K. E. Kinnear, Jr., ed. *Advances in Genetic Programming*, chapter 9. MIT Press.

Whigham, P. A. (1995) Inductive bias and genetic programming. In A. M. S. Zalzala, ed. *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 461–466, London, UK. IEE.