Data Structures and Genetic Programming

W. B. Langdon
Dept. of Computer Science,
University College London

1 Introduction

Much published work on genetic programming evolves functions without "side-effects" to learn patterns in test data, once produced they can be used to make predictions regarding new 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.* In both normal and genetic programming considerable benefits have been found in adopting a "structured approach". For example, Koza [Koz94] has shown the introduction of evolvable code modules (ADFs) can greatly reduce the effort required for GP to reach a solution.

Teller [Tel94] has shown that the introduction of indexed memory into genetic programming makes it Turing Complete. This work builds on Teller's and considers the introduction of *data structures* within GP. The expectation is that the use of data structures will prove as useful to GP as code structuring has already been shown to be. So far the important result is that GP can evolve simple data structures, such as stacks and queues [Lan95]. This abstract reports the successful evolution a list data structure, compares the use of Pareto selection and demes and describes the directed choice of crossover points.

2 Evolving a list

Aho et al's [AHU87] definition of a list gives ten list operations (Makenull, Retrieve, Insert, Delete, End, First, Next, Previous, Locate and Printlist). In the GP each operation is implemented as a separate evolving tree. Each member of the GP population contains all ten operations, plus ADFs. The first ADF takes two arguments, an integer and a function. The function is another ADF, each calling operation passes its own function. The four simpler operations (End, First, Next and Previous) can be called by the others and the ADFs (n.b. recursive calls are forbidden).

The functions and terminals used the stack and the queue were augmented with: looping constructs, a means to pass functions to ADFs, a function to swap memory locations and a print function.

Each of the ten operations represents a separate objective. The GP's task is to simultaneously optimize all of them. The value of each objective (i.e. the fitness) is determined by running each individual on 21 fixed test sequences containing 538 operations. Tests are grouped into subsequences which call several operations and cross check the values returned by them. If the checks are passed the score for each operation called in the subsequence is incremented. Only if all the checks are passed is the next subsequence started.

In addition to the ten per operation scores, an individual's fitness contains penalties for excessive CPU and memory usage. Pareto multi-objective tournament selection with niching is used to select individuals for reproduction and removal from the population.

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. On continuing the evolutionary process, solutions with reduced CPU cost were found. The three data structures, stack, queue and list and the relative difficulty of evolving them using GP, including their fitness functions, will be discussed.

```
makenul = (PROG2 (Set_Aux1 (PROG2 (SUB 1 End) (Set_Aux1 0))) 1)
retriev = (read arg1)
insert  = (write (SUB (Next aux1) (adf1 ARG2)) (write ARG2 ARG1))
delete  = (SUB (Prev aux1) (adf1 ARG1))
end     = (ADD aux1 1)
first   = 1
next    = (ADD 1 arg1)
prev    = (SUB (ADD arg1 arg1) (ADD arg1 1))
locate  = (adf1 First)
prtlist = (adf1 1)
adf1    = (SUB (forwhile (ADD 0 arg1) (forwhile aux1 aux1 0) (FUNC i0)) (ADD 0 0))
ins_adf = (swap arg1 max)
del_adf = (swap (Next arg1) arg1)
loc_adf = (ADD (SUB (read arg1) (ADD ARG1 arg1)) arg1)
prt_adf = (ADD arg1 (print (read arg1)))
```

Evolved List

3 Pareto Optimality v. Demes

Where a Pareto tournament fails to find a single non-dominated winner, it is agumented by comparing the remaining non-dominated candidates with a random sample (of up to 81) other members of the population. The one which is dominated by (or has the same score as) the least number of others is chosen.

Comparison with the rest of the population introduces a selection pressure to be different and spreads the population out over fitness space. Typically, in a population of 10,000, there are a few hundred different non-dominated fitness values (or *niches*) in the population. In contrast, when this spreading presure is absent even small demes are unable to prevent the population converging. Typically the number of occupied niches falls below 20 within the first 10 generations.

Demes have the potential advantage of breading similar programs with each other. The implementation of fitness niching used to evolve the list has exactly the opposite effect, most crossovers are between programs with different fitness'. It is not clear which is to be preferred, however mate selection could be designed to enhance the chance of breeding like with like.

4 CPU Penalty

Programs whose fitness testing requires more than a threshold number of instructions are penalized. This scheme has the anticipated benefit of curbing program run time and also causes the evolution of near parsimonious code. 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.

5 Directed Crossover

In 90% of crossovers the parents fitness and execution path is used to bias the choice of crossover location. The location is chosen to avoid disrupting code that is working, to avoid wasting crossovers by changing code that is never executed and is biased in favour of changing code that appears to be performing poorly. This appears to be beneficial but it is too early to draw firm conclusions.

6 Implementation Issues

Partial scores for each test sequence are saved. Where the location of the crossover point indicates that a child's score on a test sequence must be identical to its parent, the tests are not run, instead the partial score is used. This produces about a 50% reduction in elapse run time, however at the cost of increasing the memory required by $\approx 30\%$.

Where adfs or operations do not to have side-effects, there is no need to evaluate them more than once for each input value. Using caches to avoid such repeated evaluation reduces runtime by $\approx 50\%$.

7 Further Work

Having evolved three data structures using genetic programming, it is intended to investigate solving a number of simple problems which require the use of memory to see how helpful data *structures* are.

Whilst this work has shown fitness niches, CPU penalties and directed choice of crossover points were effective when evolving a list, further work is required to demonstrate to what extent they are generally useful. We are also investigating simple theoretical models of genetic programming.

References

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

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

[Lan95] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, San Francisco, CA., USA, July 1995. Morgan Kaufmann. To appear.

[Tel94] Astro Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1. IEEE Press, Jun 1994.