Adaptive Control of Partial Functions in Genetic Programming

Daryl Essam and R. I. (Bob) McKay
School of Computer Science,
University College, University of New South Wales, ADFA,
Northcott Drive, Campbell, ACT,
Canberra, Australia
{daryl,rim}@cs.adfa.edu.au

Abstract- This paper investigates the use of partial functions in genetic programming. Previous work has shown that the convergent behaviour of populations of partial functions is very similar to that of populations of total functions. However the convergence rates of populations of partial functions have been slower. The results presented here demonstrate a significant improvement in the rate of convergence of populations of partial functions, and indicate that partial functions represent a realistic alternative to total functions for a range of problems.

Keywords: Genetic Programming, Partial Functions, Fitness Evaluation

1 Introduction

Historically, practitioners of genetic programming have emphasised the need for complete definition of the programs being evolved. Hence early work by Koza(1992) and others emphasised the importance of the completeness of the operators involved, while approaches such as Strongly Typed Genetic Programming (Montana 1995) provided mechanisms to ensure totality of programs while permitting individual operators to be defined only for a subset of all potential inputs.

This emphasis on the totality of the functions evolved acts as a straightjacket on genetic programming, limiting the algorithms and data structures investigated – algorithms, operators or data structures which do not preserve totality are likely to be dismissed from further investigation. Previous work (McKay 2000a,b) on two toy and one real-world GP applications have suggested that the converged behaviour of populations of partial and of total functions are very similar. However with the algorithms used there, populations of partial functions reach convergence more slowly than do populations of total functions, particularly when measured in terms of generations, though the difference is reduced when measured in cpu utilisation. This paper investigates an adaptive approach which leads to faster convergence of populations of partial programs. Preliminary results show a significantly increased rate of convergence, with only small differences remaining between the cpu cost of populations of partial and of total functions.

This is a self-archived copy of the accepted paper, self-archived un- der IEEE policy. The authoritative, published version can be found at http://ieeexplore.ieee.org/xpls/abs all.jsp?arnumber=934285&tag=1

1.1 Why partial functions

The original justification for this work lay in the hope that partial functions would permit greater population diversity and hence permit superior performance at convergence. In fact, this hope has not been borne out by experience – the converged behaviour of populations of partial and of total functions have turned out to be remarkably similar. In consequence, there is at least no convergent-behaviour dis-advantage in using populations of partial functions.

However in the course of our investigations, it has become clear that new and interesting algorithms for genetic programming become available when the restriction to total functions is relaxed; we hope to be able to demonstrate some examples in the near future.

The aim of this paper is to further investigate the computational cost of using populations of partial functions. Earlier work has suggested a small convergence rate disadvantage to the use of populations of partial functions. This paper investigates an adaptive variant of the previously proposed fixed algorithm, which promises to significantly reduce computational cost.

2 Details of Approach

As with the previous work, comparisons are based on two classes of problem: learning (recursive) list membership in a lisp-like language, and learning boolean descriptions for 6- and 11-multiplexers. These problems are described in more detail below. Earlier work had established that, on these problems, early use of implicit fitness sharing (Deb and Goldberg 1989, Smith et al 1992) led to improved performance. The experiments therefore all use fitness sharing in the initial stages. The experiments compare the performance of populations of total functions with that of populations of partial functions using an adaptive fitness measure described in more detail below.

2.1 Partial Functions

A partial function is a function whose value is not defined for some argument values. In these experiments, partial functions are represented by the use of a special symbol, 'undef', which may occur at any point in the program tree. When the program runs, any evaluation for which 'undef' is an argument evaluates to 'undef' unless the value of the function is independent of that argument. For example, boolean 'and' has the truth table shown in table 1.

The system is based on Ross' (1999) DCTG-GP system. DCTG-GP was used because its explicit representation of the syntax and semantics of the program populations provided a simple mechanism to specify the syntax and semantics of the 'undef' symbol. However the grammars used merely encode the typing of the problem space, and so the results apply not only to grammar-guided genetic programming (Whigham 1995), but extend to strongly typed genetic programming (Montana 1995).

and	**false**	**undef**	**true**
false	false	false	false
undef	false	undef	undef
true	false	undef	true

Table 1: Truth Table for Boolean and

2.2 Implicit Fitness Sharing and Partial Functions

The approach to fitness sharing used is described in detail in McKay 2000a. Briefly, we replace the raw fitness function for an individual i

$$f_{raw}(i) = \sum_{c \in cases} reward(i(c))$$

with the shared fitness function

$$f_{share}(i) = \sum_{c \in cases} \frac{reward(i(c))}{\sum_{i':i'(c)=i(c)} reward(i'(c))}$$

With populations of partial functions, another issue arises: if the payoffs from the sub-problems are simply added together (assuming there are no negative payoffs), then there is evolutionary pressure toward totality, because even small rewards for poor predictions are better than no reward at all. This pressure would defeat the intention, which is to permit increased diversity through partial functions. Hence in this work, the shared fitnesses are divided by the number of sub-problems which the program attempts to solve (that is, the fitness of an individual program is the mean of the shared rewards it receives, averaged over all the sub-problems for which its answer is not 'undef'). For an individual i, let N(i) be the number of cases c for which i(c) is defined, then:

$$f_{part_share}(i) = \sum_{c \in cases} \frac{reward(i(c))}{N(i) * \sum_{i':i'(c)=i(c)} reward(i'(c))}$$

Thus the evolutionary pressure on partial functions is toward accuracy on sub-problems. This nay be seen by considering two individuals that correctly solve the same set of sub-problems. The individual that has the smaller *N(I)* , will have the largest fitness. Consider also, an individual that solves only one problem, but does so correctly - that individual may have an equivalent fitness as another individual which solves many problems, where again they are all solved correctly. Thus, this pressure will not necessarily result in individuals capable of solving the whole problem.

There are many possible mechanisms to alleviate this. Previously (McKay 2000c) we have used ensemble learning mechanisms, unfortunately with only limited success. An alternative approach is to use the partial function fitness sharing measure defined above early in the run, but to switch to raw fitness at the end of the run. Our previous work used a fixed, ramped schedule for this purpose. Unfortunately the partial share fitness metric above appears to be prone to over-fitting: detailed analysis of the populations showed many individuals which were defined for just one or two learning instances. A fixed schedule permits this over-fitting to continue, to the detriment of the learning success. In this paper, we use instead an adaptive approach, in which the gradual change from partial function fitness sharing to raw fitness depends on statistics from the population. The statistics used depend intrinsically on the partial nature of the functions in the population. Thus it is not possible to undertake a similar approach for total functions. Instead, for comparison purposes we use the most similar algorithm we have available, namely the fixed ramped approach used in previous work.

2.3 Stepped Evaluation

Stepped evaluation modifies the partial evaluation function to the following:

$$f_{stepped}(i) = \sum_{c \in cases} \frac{reward(i(c))}{\max(N(i),k) * \sum_{i':i'(c)=i(c)} reward(i'(c))}$$

This equation incorporates the new term, *max(N(I), k)*. *k* is a variable that gradually increases from the initial value of 1. Considering *k*, if it has the value 1, then the above evaluation is identical to the previous partial/share evaluation. If *k* equals the number of cases, then (modulo a constant) we get the standard fitness sharing formula. In stepped evaluation, the value of *k* changes through the evaluation, stepping gradually upwards from 1. The key decision is the rate at which *k* changes. Two user-definable parameters, α and β, are used to control this.

For each generation, the number of individuals that solve *k* cases is evaluated. If that number exceeds $\alpha\%$, then *k* is increased by $\beta\%$. Thus the values of α and β are critical; when α is small, *k* will often be increased; when β is large, the increases will also be large. Ideally, *k* should advance at a rate appropriate to the condition of the individuals, gradually providing incentive for them to cover more cases, whilst gradually removing narrowly focused individuals.

The search space for this experiment is defined by the grammar in table 2 (for total functions, the productions leading to 'undef' are deleted):

3 Experiments: List Membership

```
S -> M
M -> if EXPN EXPN M
M -> ''
M -> undef
EXPN -> atom LST
EXPN -> eq LST LST
EXPN -> member x LST
EXPN -> true
EXPN -> false
EXPN -> undef
LST -> first LST
LST -> rest LST
LST -> x
LST -> y
LST -> undef
```

Table 2: Grammar for List Membership

The recursive call to member allows the possibility of infinite loops. To prevent this, a count of the depth of looping was kept, and a depth greater than 20 caused the function to return the value 'loop'; this was treated in fitness evaluation as an incorrect (but defined) answer.
The examples for learning this function consisted of ten true cases and ten false, and are shown in table 3.

TRUE CASES	FALSE CASES
member(1 [1])	member(1 [6])
member(1 [2 1])	member(1 [3 6])
member(1 [2 3 1])	member(1 [2 3 6])
member(1 [2 3 4 1])	member(1 [2 3 4 6])
member(1 [2 3 4 5 1])	member(1 [2 3 4 5 6])
member(1 [2 3 4 5 6 1])	member(1 [2 3 4 5 6 7])
member(1 [2 3 4 5 6 7 1])	member(1 [2 3 4 5 6 7 8])
member(1 [2 3 4 5 6 7 8 1])	member(1 [2 3 4 5 6 7 8 9])
member(1 [2 3 4 5 6 7 8 9 1])	member(1 [2 3 4 5 6 7 8 9 2])
member(1 [2 3 4 5 6 7 8 9 2 1])	member(1 [2 3 4 5 6 7 8 9 2 3])

Table 3: List Membership Cases

The aim of the experiment was to find a program which correctly computes membership. An example solution is:

```
(if (eq x (first y))
    true
    (if (member x (rest y))
        true
        false))
```

The experimental setup used tournament selection and half-ramped initialisation as implemented in DCTG-GP (Ross, 1999); table 4 shows experimental parameters:

PARAMETER	SPECIFICATION
Number of Runs	100
Max Generations	200
Population Size	500
Max depth (initial pop)	8
Max depth (subsequent)	10
Tournament size	5
Crossover Probability	0.9
Mutation Probability	0.1

Table 4: Run Parameters (List Membership)

The raw fitness function was the proportion of the 20 cases correctly solved. In principle, it would be possible for a non-recursive program to solve all 20 cases, but not within the maximum depth imposed on the population. Each run terminated at 200 generations.

4 Results: List Membership

Figure 1 shows the percentage of runs incomplete plotted against generation. In all plots, the legend shows the α value of the partial function runs first, then the β value (ie partial, 50%: 20% is a run with a population of partial functions, with α = 50% and β = 20%).

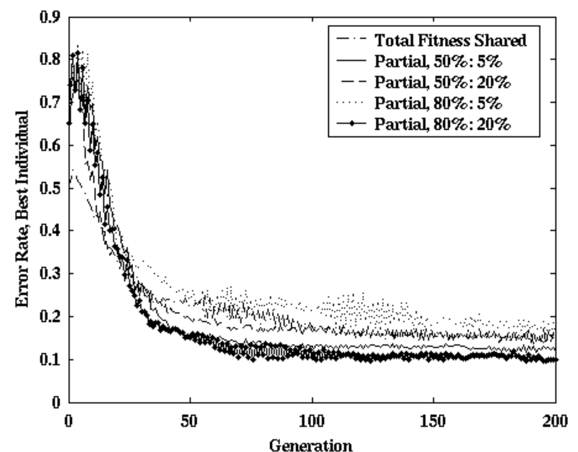

Figure 1: List Membership, Error rate of Best Individual by Generation

Our experience with populations of partial functions is that they take less time to evaluate a generation than with total functions. Hence figure 2 shows the same data plotted against cpu time.

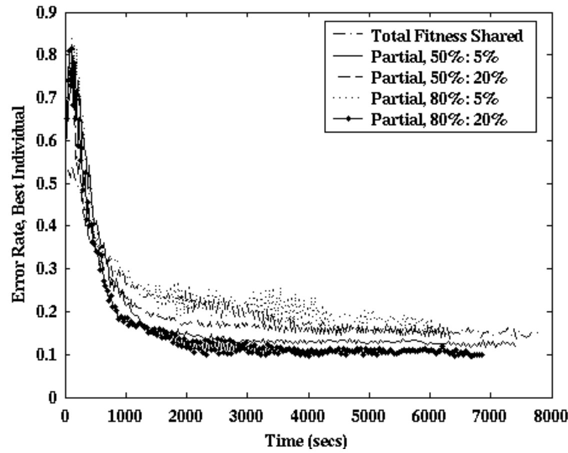

Figure 2: List Membership, Error rate of Best Individual by cpu time

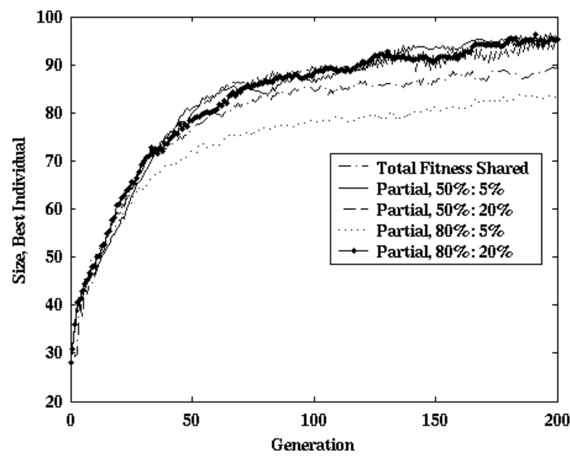

Figure 3: List Membership, Average Size of Individuals

From figure 3, we might hypothesise that the relative performance time/generation performance is simply an artefact of the size of the programs generated.

5 Experiments: Multiplexers

Two sets of experiments were conducted, using the 11-multiplexer and 6-multiplexer problems. The former seeks a boolean expression for a multiplexer with two address and four data lines, the latter with three address and eight data lines. The search space for the 6-multiplexer is defined by the grammar in table 5 (for total functions, the productions leading to 'undef' are deleted).

EXPR → BOOL
BOOL → TERM
BOOL → and BOOL BOOL
BOOL → or BOOL BOOL
BOOL → not BOOL
BOOL → if BOOL BOOL BOOL
BOOL → undef
TERM → a0
TERM → a1
TERM → d0
TERM → d1
TERM → d2
TERM → d3
TERM → undef

Table 5: Grammar for 6-Multiplexer

The search space for the 11 multiplexer extends this by adding TERM productions for address line a2 and data lines d4 through d7.

The examples for learning the 6 multiplexer consisted of the 64 possible input/output pairs - see table 8. For the 11 multiplexer, computational cost precluded evaluation over the 2048 input/output pairs in each generation. Instead, for each generation, 64 of these pairs were randomly selected and used to evaluate that generation. Since termination was based on a zero error rate for these 64 cases, it is possible that some incorrect solutions were accepted as correct. However this possibility does not affect the comparisons undertaken in this work, since all treatments are affected equally.

PARAMETER	SPECIFICATION
Number of Runs	100
Max Generations	500 (11 multiplexer)
	200 (6 multiplexer)
Population Size	150
Max depth (initial pop)	8
Max depth (subsequent)	10
Tournament size	5
Crossover Probability	0.9
Mutation Probability	0.1

Table 6: Run Parameters (Multiplexers)

The experimental setup used tournament selection and half-ramped initialisation; experimental parameters for the 11 and 6 multiplexer runs are given in tables 6 and 7:

6 Results: Multiplexers

The Error rate by generation for the 11 multiplexer is shown in figure 4; figure 5 shows the same result by cpu time, while figure 6 shows the evolution of program size. Again, the results are consistent with the hypothesis that the cause of the faster execution of the partial function populations is simply the smaller size of the partial functions. However this hypothesis is not compatible with figures 7, 8 and 9, showing the equivalent results for the 6 multplexer. The cause of the faster execution of partial function populations thus requires further investigation.

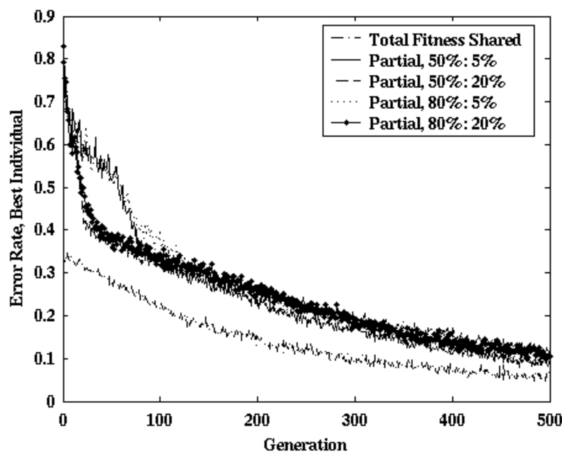

Figure 4: 11 Multiplexer, Error rate of Best Individual by Generation

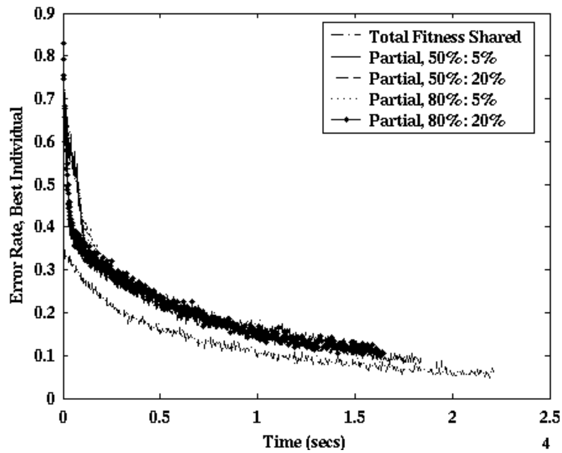

Figure 5: 11 Multiplexer, Error rate of Best Individual by cpu time

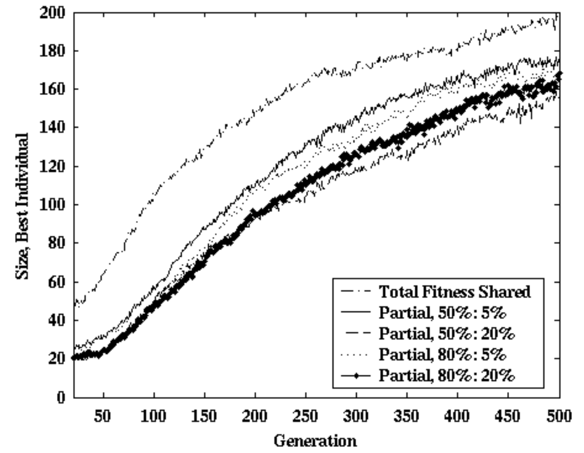

Figure 6: 11 Multiplexer, Average Size of Individuals

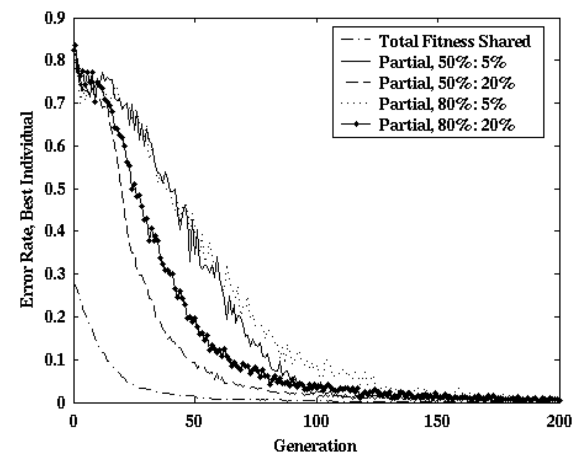

Figure 7: 6 Multiplexer, Error rate of Best Individual by Generation

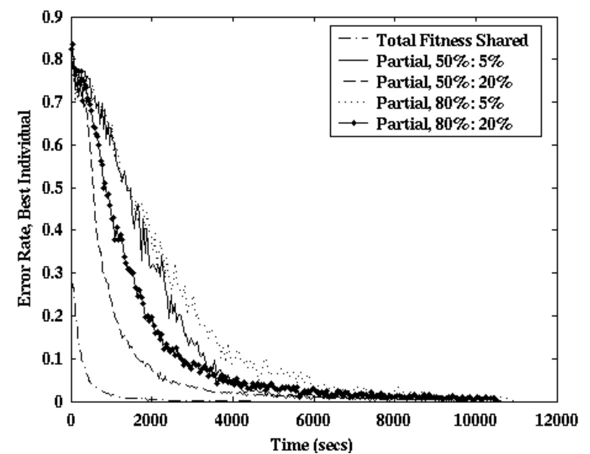

Figure 8: 6 Multiplexer, Error rate of Best Individual by cpu time

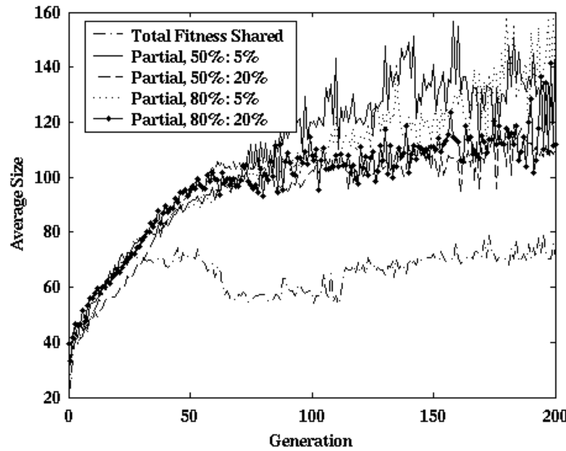

Figure 9: 6 Multiplexer, Average Size of Individuals

7 Discussion

The results presented here are consistent with our previous results, showing that populations of partial functions have similar converged behaviour to populations of total functions. However with the right choice of parameters, stepped evaluation now gives convergence rates for partial functions very similar to those obtained from total functions, both in terms of number of evaluations, and particularly in terms of computation time. The one exception to this is the very simple 6-multiplexer problem. Perhaps the issue here is that the problem is so simple that the stepped evaluation approach does not have time to act before the total functions population has converged.

In all three experiments, the best performance for partial functions was achieved by the α = 50% and β = 20% settings. Interestingly, this is the most eager of the four settings – with this set of parameters, the system is more often prepared to alter k, and to alter it in larger steps.

The four sets of α and β were chosen by informed guesswork. The results would suggest that it may be worth investigating still more eager settings of the algorithm – ie smaller values of α and large values of β. With the appropriate choices of α and β, it is conceivable that stepped evaluation with partial functions may even outperform the convergence rate of total functions, at least in terms of cpu time, if not number of evaluations.

The experiments reported here used a range of parameter values chosen more-or-less at random. It seems apparent that with appropriate choice of parameter settings, the performance of partial functions could be brought even closer to, and perhaps exceed, the performance of total functions. An alternative approach is to try to eliminate the parameters. After conversations with Xin Yao, we are now planning to investigate the possibility of replacing fitness sharing by an anti-correlation metric, further enhanced by the analysis of information gain.

8 Conclusions

The results presented here are at least suggestive that, combined with suitable fitness evaluation mechanisms, populations of partial functions can perform at a comparable level with populations of total functions. There is still a clear need for further work comparing populations of partial and total functions on a much wider range of problems, and we plan to undertake that in the relatively near future.

The most important consequence of this work is the possibility that genetic programming can escape the restriction to populations of total functions. This greatly widens the pool of potential algorithms and problem representations available. We hope to present some new algorithms based on this idea in the near future.

Acknowledgements

The ideas in this paper have benefited greatly from discussions over the years with Paul Darwen, Peter Whigham, Xin Yao, Ko-Hsin Liang and Hussein Abbass. Thank you.

Bibliography

Deb, K and Goldberg, D E: 'An investigation of niche and species formation in genetic function optimization' in J D Schaffer (Ed) *Proceedings of the Third International Conference on Genetic Algorithms*, Pp 42-50, Morgan Kaufmann, 1989

Koza, J R 'Genetic Programming: on the Programming of Computers by means of Natural Selection', Bradford / MIT Press, 1992

McKay, R I: 'Partial Functions in Fitness-Shared Genetic Programming', *Proceedings of the 2000 Congress on Evolutionary Computation*, IEEE Piscataway, 2000, Pp 349 – 356

McKay, R I: 'Variants of Genetic Programming for Species Distribution Modelling – Fitness Sharing, Partial Functions, Population Evaluation', submitted to *Evolutionary Computation*; abstract in Recknagel, F (ed) *Abstracts of the Second International Conference on Applications of Machine Learning to Ecological Modelling*, Adelaide, November 2000

McKay, R I: 'Committee Learning of Partial Functions in Fitness-Shared Genetic Programming', CD-ROM *Proceedings of the 2000 Conference of the IEEE Industrial Electronics Society* and *Simulated Evolution and Learning 2000*, IEEE Press, Piscataway, 2000

Montana, D J: 'Strongly Typed Genetic Programming', *Evolutionary Computation* 3(2), Pp. 199-230, 1995

Ross, B J: 'Logic-based Genetic Programming with Definite Clause Translation Grammars', Technical Report

#CS-99-02, Dept of Computer Science, Brock University, St Catharines Ontario, 1999; summary in Banzhaf et al (eds) *Proceedings of the Genetic and Evolutionary Computation Conference*, P1236, Morgan Kaufmann, 1999

Smith, R E, Forrest, S and Perelson, A S: 'Searching for diverse, cooperative populations with genetic algorithms', *Evolutionary Computation* 1(2), Pp 127 - 149, 1992

Whigham, P A: 'Grammatically-biased Genetic Programming' in J Rosca (ed) *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Pp 33-41, Morgan Kaufmann, 1995