Partial Functions in Fitness-Shared Genetic Programming

R. I. McKay
School of Computer Science, Australian Defence Force Academy,
Northcott Drive, Campbell, ACT,
Canberra, Australia
rim@cs.adfa.edu.au

Abstract- This paper investigates the use of partial functions and fitness sharing in genetic programming. Fitness sharing is applied to populations of either partial or total functions and the results compared. Applications to two classes of problem are investigated: learning multiplexer definitions, and learning (recursive) list membership functions. In both cases, fitness sharing approaches outperform the use of raw fitness, by generating more accurate solutions with the same population parameters. On the list membership problem, variants using fitness sharing on populations of partial functions outperform variants using total functions, whereas populations of total functions give the best performance on multiplexer problems.

1 Introduction

Genetic programming, like other forms of evolutionary computation, can suffer from premature convergence, whereby variation is eliminated from a population before the desired problem solution is achieved. Previous work (McKay 2000) has demonstrated that fitness sharing, widely used in other forms of evolutionary computation to increase diversity and delay convergence, can lead to better performance - measured in terms of error rate - in genetic programming as well.

Fitness sharing was introduced by Deb and Goldberg (1989), in a form (explicit fitness sharing) which relied on a distance metric to define the similarity between individuals. Similar individuals are punished for that similarity by being required to share the raw fitness they receive. However there are disadvantages to this approach, because it requires a priori definition of the distance function, before knowledge of the shape of the search space has been acquired.

Smith, Forrest and Perlson (1992) noted that the requirement to define a distance metric could be avoided in problems where the fitness of an individual is built up from the payoff from discrete sub-problems. In this case, the payoff for a sub-problem could simply be shared amongst the individuals which perform well on that sub-problem. The resultant approach is known as implicit fitness sharing. Since a high proportion of genetic programming problems share this payoff structure, implicit fitness sharing is very relevant.

Most work in evolutionary computation makes use of total functions. In many cases this is a matter of simple necessity - for example, the genome structure in standard genetic algorithms does not support partially defined functions ('don't care' symbols simply indicate that a particular input value is ignored, not that the output value is undefined for a given input). But partial functions are rarely, if ever, used in genetic programming, where they do have a sensible meaning. Presumably this arises from the argument for implicit parallelism - if implicit parallelism is a good thing, then the more of it the better; but partial functions forego some implicit parallelism by not attempting to solve some parts of a problem.

On the other hand, total functions are under evolutionary pressure to solve all parts of a problem; this pressure tends to reduce diversity, perhaps fatally so if a critical part of an optimal solution is more difficult to find than equivalent parts of a local optimum.

The aim of the work begun here is to investigate the tradeoffs between the greater implicit parallelism of total functions, and the potentially greater diversity provided by partial functions, and their interaction with the diversity-promoting mechanism of implicit fitness sharing.

2 Details of Approach

Comparisons have been carried out 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. The experiments compare the use of implicit fitness sharing, raw fitness, and a combination of the two. Each treatment is repeated twice, using a population of total functions, and a population of partial functions.

2.1 Partial Functions

In these experiments, partial functions are represented by the use of a distinguished symbol, 'undef', which may be inserted at any point in the program tree. When the program is run, any function evaluation for which 'undef' is an argument evaluates to 'undef' unless the value of the function is independent of that argument. For example, the boolean 'and' has the truth table shown in table 1:

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

Table 1: Truth Table for Boolean and

The system has been based on Ross' (1999) innovative 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 simply 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 1994).

2.2 Implicit Fitness Sharing and Partial Functions

Fitness sharing aims to reduce the eagerness of evolutionary search and encourage diversity by providing a reward for diversity. Thus fitness sharing algorithms typically reduce the early performance of an algorithm, but more than compensate by delaying convergence, and producing better asymptotic behaviour.

For populations of total functions, the implementation of implicit fitness sharing is straightforward. The payoff for a particular sub-problem is shared amongst all population members which correctly answer the sub-problem (in the problems used here, values are discrete, so grades of correctness do not need to be considered).

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 which are free to concentrate on particular sub-problems. 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').

Thus the evolutionary pressure on partial functions is toward accuracy on sub-problems. This pressure will not necessarily result in individuals capable of solving the whole problem. There are many possible mechanisms to alleviate this, ensemble learning mechanisms being particularly notable. However a very simple approach has been taken in this paper. In addition to runs using raw fitness throughout, and others using shared fitness throughout, there is a third set of runs, in which the fitness measure changes from shared fitness to raw fitness using a fixed schedule. The particular schedule was chosen a priori. It uses fitness sharing for the first 25% of generations of a run, and raw fitness for the last 25%, with a linear ramp between fitness sharing and raw fitness for the remaining 50% of generations (the raw and shared fitnesses are normalised to have the same mean before being added together)

This schedule is doubtless not optimal - the optimal schedule is almost certainly problem-dependent. But most obvious mechanisms to optimise the schedule lead to problems of fairness of comparison with the other two treatments.

3 Experiments: List Membership

The search space for this experiment (derived from Whigham 1996) is defined by the grammar in table 2 (for total functions, the three productions leading to 'undef' are deleted):

```
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; experimental parameters are given in table 4:

PARAMETER	SPECIFICATION
Number of Runs	100
Max Generations	200
Population Size	1000
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 used was the proportion of the twenty cases correctly solved. In principle, it would be possible for a non-recursive program to solve all twenty cases, but not within the maximum depth imposed on the population.

Each run was terminated at 200 generations, or earlier if it found a correct solution to the problem.

4 Results: List Membership

The percentage of runs which terminated in a correct solution are shown in table 5:

	Total Funcs	Partial Funcs
Raw Fitness	63	43
Ramped Fitness	79	84
Shared Fitness	79	88

Table 5: Percentage of 100 Runs Generating Correct Solution

The differences between the raw fitness and other treatments are significant at the 1% level (maximum likelihood ratio test) for both the total and partial function cases. The other differences are of lower significance, but the difference between the total and partial functions, when the ramped and shared fitness cases are aggregated together, gives a probability value of 6.5% for the null hypothesis. To confirm this, the trials involving fitness sharing were replicated for a further 200 runs. The results are shown in table 6:

	Total Funcs	Partial Funcs
Ramped Fitness	79.3	84.7
Shared Fitness	82.7	88

Table 6: Percentage of 300 Runs Generating Correct Solution

With a total of 300 runs, the null hypothesis for the difference between partial and total functions has a probability value of 1.2%, when ramped and shared fitness are considered together. Considered separately, the null hypothesis probabilities are 8.8% and 6.4% respectively. The differences between ramped and shared fitness are of low significance, even when total and partial results are considered together.

Figure 1 shows the percentage of runs incomplete plotted against generation. In the legend, 'total' and 'partial' refer to populations of partial and total functions respectively, and 'raw', 'share' and 'ramp' refer to the use of raw fitness throughout a run, fitness sharing throughout a run, and the ramped approach described above (to improve readability, the partial functions/raw fitness treatment is omitted in all figures since it is of little independent interest).

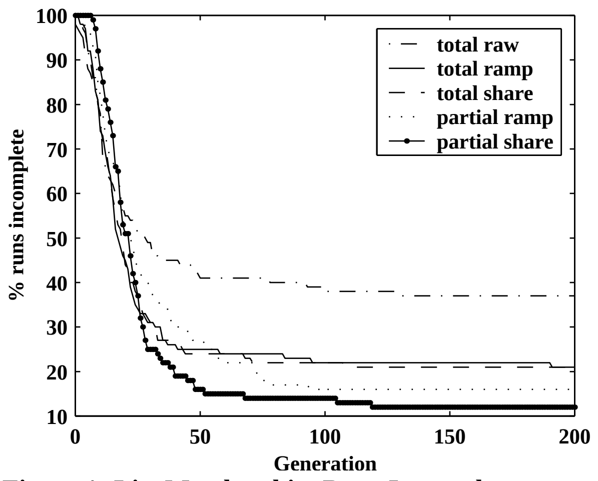

Figure 1: List Membership, Runs Incomplete

Figure 2 shows the error rate of the fittest individual plotted against generation (the first 20 generations are omitted for clarity

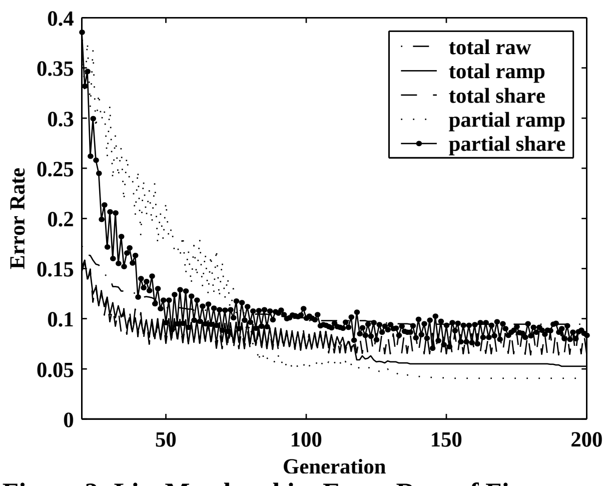

Figure 2: List Membership, Error Rate of Fittest Individual

One interesting aspect of figure 2 is the 2-generation oscillation in error rate exhibited by all fitness-sharing treatments. On detailed examination, the populations in many runs contain a group of individuals with low error rate (and hence high raw fitness), and another group of very different individuals with higher error rate, but covering a different subset of the test cases. Small fluctuations in the size of the second set are reflected in larger changes in the shared fitness of its members relative to the first set, and hence to the selection pressure toward it, resulting in the oscillations seen. These fluctuations can also be seen in figure 3, which shows the standard deviation of the number of individuals in the population covering each test case, and hence is a proxy measure of the phenotypic diversity of the population (if the population is of low diversity, then some test cases will be well covered, and others poorly, so the variance in cover will be high).

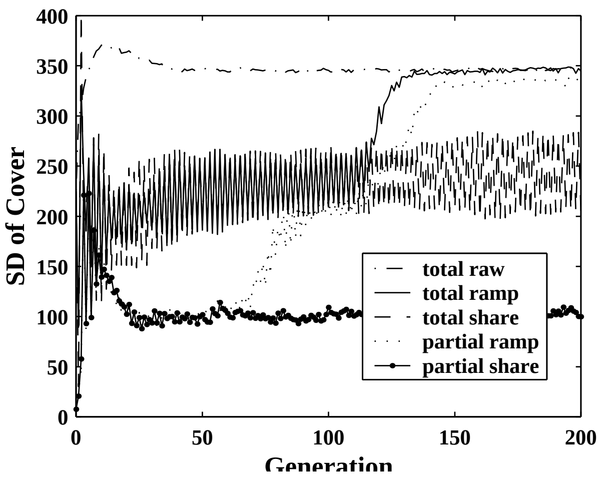

Figure 3: List Membership, Standard Deviation of Cover of Test Cases

Population convergence is strongly associated with the phenomenon of bloat (Nordin et al, 1994). If fitness sharing acts to delay convergence, then a consequent reduction in bloat might occur. Figure 4 shows the average tree depth in the population (for clarity, the plot begins at generation 20). There appears to be a small reduction in tree depth associated with fitness sharing of total functions, and a clear reduction with fitness sharing of partial functions.

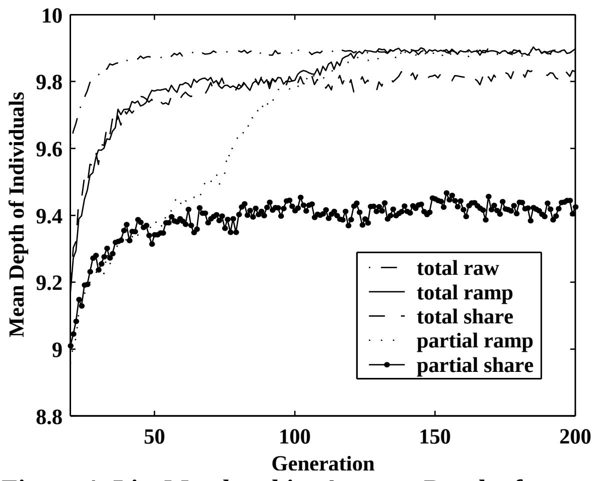

Figure 4: List Membership, Average Depth of Program Tree

5 Experiments: Multiplexers

Two sets of experiments were conducted, using the 6-multiplexer and 11-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 7 (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 7: 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.

Inputs						Output
a0	a1	d0	d1	d2	d3	
0	0	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	1	0	0	1
1	1	1	1	1	1	1

Table 8: 6 Multiplexer Input/Output Examples

The aim of the experiment was to find a boolean function which correctly defines the multiplexer. An example solution for the 6-multiplexer is:

```
(if a0 (if a1 d3 d2)
       (if a1 d1 d0))
```

The experimental setup used tournament selection and half-ramped initialisation; experimental parameters are given in table 9:

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

Table 9: Run Parameters (Multiplexer)

Each run of the 6-multiplexer was terminated at 100 generations, or earlier if it found a correct solution to the problem. The 11-multiplexer runs were terminated at 200 generations, or earlier on a correct solution.

As with the membership problem, three forms of fitness evaluation were used: raw fitness, implicit fitness sharing, and the ramped approach previously described. Each form was evaluated both on populations of total functions, and on populations of partial functions.

6 Results: Multiplexers

The percentage of runs which terminated in a correct solution are shown in tables 10 and 11:

	Total Funcs	Partial Funcs
Raw Fitness	84	75
Ramped Fitness	100	88
Shared Fitness	100	82

Table 10: Percentage of Runs Generating Correct Solution (6 Multiplexer)

For the 6 multiplexer, the differences between total and partial functions are of low significance for the raw fitness treatment, but the null hypothesis has a probability of under .001% for both ramped and shared fitness. Similarly, the differences between raw fitness and ramped and shared fitness are highly significant for total functions. For partial functions, the ramped and raw fitness treatments are significantly different (p = 1.2%), but the other differences are of low significance.

	Total Funcs	Partial Funcs
Raw Fitness	17	4 (2/50)
Ramped Fitness	76	54
Shared Fitness	91	0 (0/20)

Table 11: Percentage of Runs Generating Correct Solution (11 Multiplexer)

For the 11 multiplexer, two of the experiments are incomplete (50 and 20 runs respectively completed so far), but the incomplete results are sufficient that all differences in the table, except that between raw and shared fitness for partial functions, are significant at the 1% level.

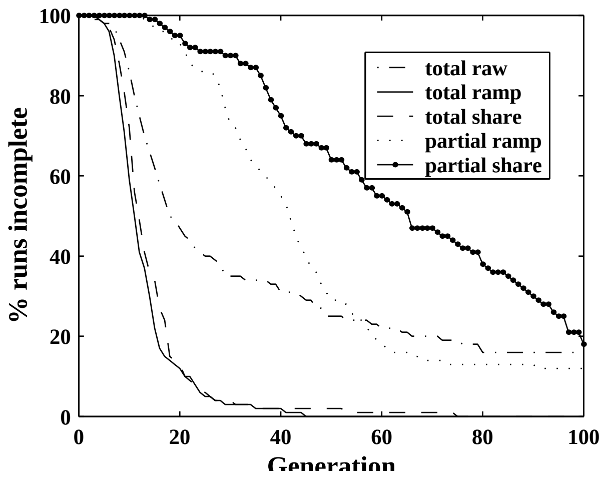

Figure 5: 6 Multiplexer, Runs Incomplete

Figure 5 shows the percentage of incomplete runs as a function of generation for the 6 multiplexer. It is clear from this figure that the partial function treatments, especially that using shared fitness throughout are not converged by the end of 100 runs, hence they could eventually achieve performance closer to that obtained using total functions. This is supported by figure 6, showing the error rate of the best individual vs generation.

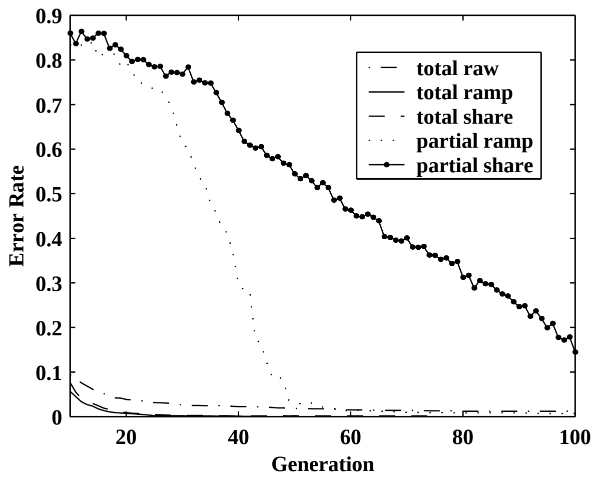

Figure 6: 6 Multiplexer, Error Rate of Fittest Individual

Similar conclusions may be drawn from figures 7 and 8, showing the percentage of incomplete runs, and the error rate of the fittest individual, for the 11 multiplexer.

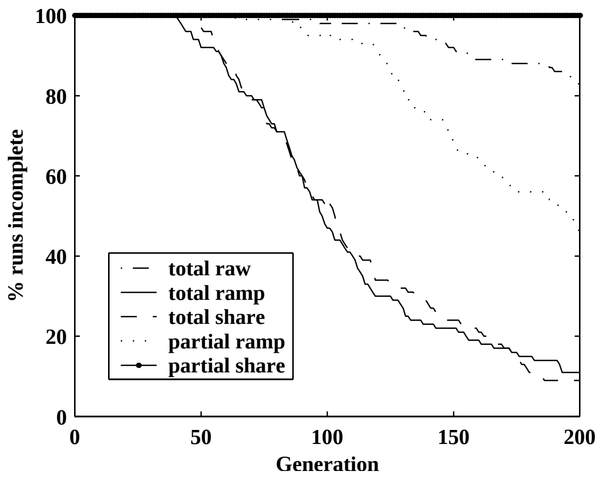

Figure 7: 11 Multiplexer, Runs Incomplete

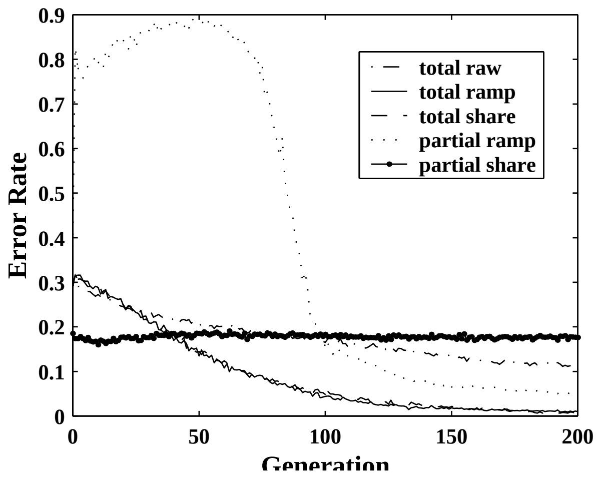

Figure 8: 11 Multiplexer, Error Rate of Fittest Individual

Figures 9 and 10 show the standard deviation of cover of the test cases for the 6- and 11-multiplexers respectively. Again, there are strong indications of the ability of fitness sharing to maintain population diversity, especially when used with partial functions.

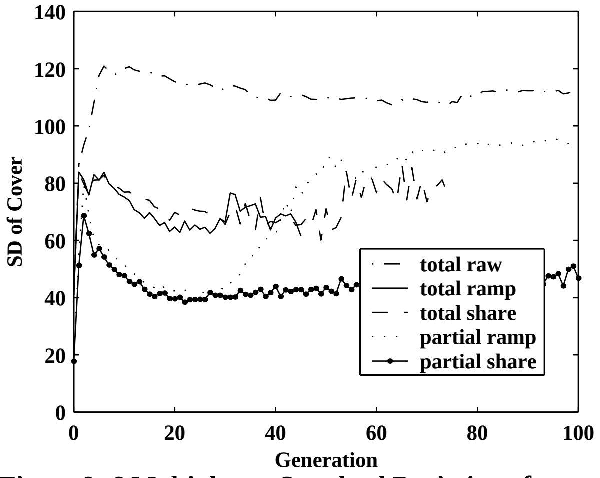

Figure 9: 6 Multiplexer, Standard Deviation of Cover of Test Cases

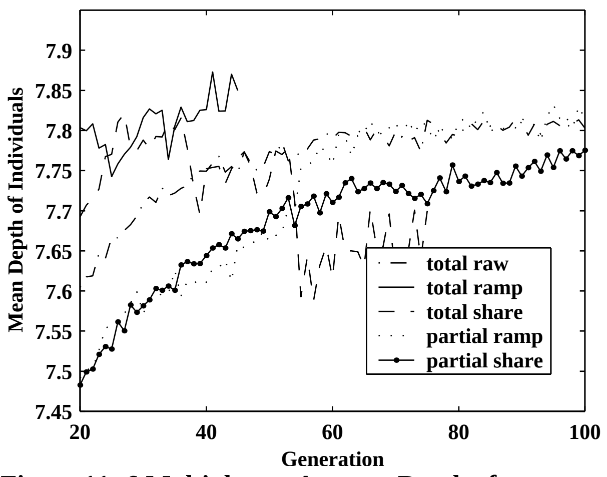

Figure 11: 6 Multiplexer: Average Depth of Program Tree

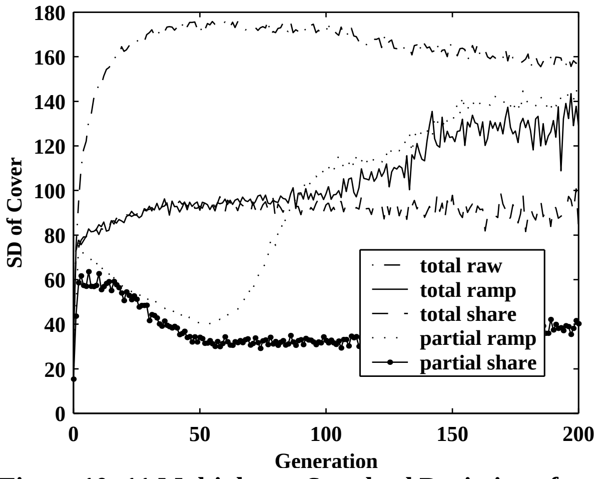

Figure 10: 11 Multiplexer, Standard Deviation of Cover of Test Cases

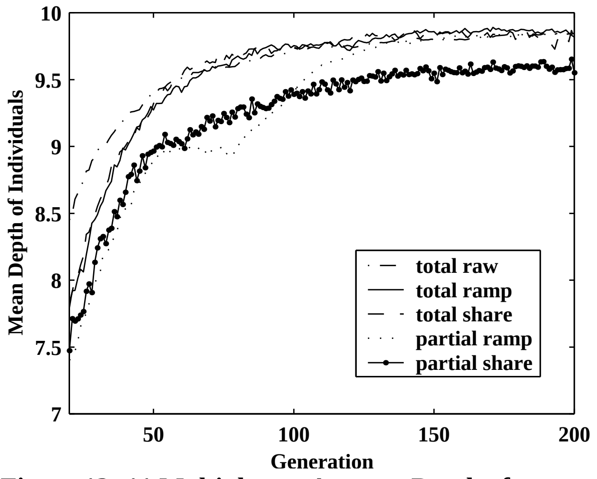

Figure 12: 11 Multiplexer: Average Depth of Program Tree

Figures 11 and 12 show the mean depths of individuals for the 6- and 11-multiplexers respectively (the first 20 generations are omitted for clarity). As with the membership problem, there is a noticeable reduction in mean depth for populations of partial functions with fitness sharing. The reduction is more clearly marked in the 11-multiplexer case.

7 Conclusions

In the experiments reported here, covering two very different types of problems, there is strong evidence of the ability of fitness sharing to maintain population diversity and delay convergence. This ability is enhanced when fitness sharing is applied to populations of partial functions rather than populations of total functions.

The delayed convergence led to considerably better performance by approaches based on fitness sharing (when compared with raw fitness) for all three problems considered. This improvement occurred whether performance was measured by percentage of runs finding correct solutions, or by error rate at convergence.

The increased delay in convergence provided by populations of partial functions led to significantly better performance over 200 generations in the recursive list membership problem, under both measures.

For the 6 multiplexer problem, the performance of fitness sharing on populations of total functions was so good that further delaying convergence through the use of partial functions could only serve to reduce performance.

In the 11 multiplexer problem, computational costs precluded following the experiments to convergence. At 200 generations, the performance of fitness-shared populations of total functions was better than that of fitness-shared populations of partial functions, but the possibility remains that the converged behaviour of populations of partial functions under ramped fitness sharing may be comparable.

There are strong indications that fitness sharing in populations of partial functions can lead to a significant reduction in bloat, by comparison with populations of total functions.

Acknowledgements

The ideas in this paper have benefited greatly from discussions over the years with Paul Darwen, Peter Whigham, Xin Yao and Ko-Hsin Liang. For assistance with the statistical discussion, I thank Ted Catchpole. The system has been implemented through modifications to Brian Ross' innovative DCTG-GP system, which contributed greatly to the rapidity with which a wide range of different algorithms could be prototyped in the development of this system, and supported a simple mechanism for defining and evaluating partial functions.

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

McKay, R I: 'Fitness Sharing in Genetic Programming', submitted to Genetic and Evolutionary Computation Conference (GECCO-2000)

Montana, D J: 'Strongly Typed Genetic Programming', Technical Report BBN 7866, Bolt Beranek and Newman, Inc, Cambridge MA, 1994

Nordin, P, Francone, F and Banzhaf, W: 'Explicitly defined introns and destructive crossover in genetic programming' in J P Rosca (ed) *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications'*, Pp 6-22, Tahoe City, Cal., 1994

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: 'Grammatical Bias for Evolutionary Learning' PhD thesis, Australian Defence Force Academy, University of New South Wales, Canberra, 1996

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