Committee Learning of Partial Functions in Fitness-Shared Genetic Programming

R I (Bob) McKay

School of Computer Science
Australian Defence Force Academy
Northcott Drive, Campbell, ACT 2600 Australia
rim@cs.adfa.edu.au

Abstract

This paper investigates the application of committee learning to fitness-shared genetic programming. Committee learning is applied to populations of either partial and total functions, and using either fitness sharing or raw fitness, giving four treatments in all. The approaches are compared on three problems, the 6- and 11-multiplexer problems, and learning recursive list membership functions. As expected, fitness sharing gave better performance on all problems than raw fitness. The comparison between populations of partial and total functions with fitness sharing is more equivocal. The results are very similar, though slightly in favour of total functions. However there are strong indications that the average size of individuals in the partial function populations are smaller, and hence might be expected to generalise better, though this was not investigated in this paper.

Index terms-- committee learning, fitness sharing, genetic programming, partial functions, population diversity

1. Introduction

Genetic programming, like other forms of evolutionary computation, can suffer from premature convergence. Fitness sharing [1] is one of a number of approaches used by the evolutionary community to maintain population diversity and delay premature convergence. A variant, implicit fitness sharing, [2] is particularly well suited to many of the forms of learning problems which arise in genetic programming.

Previous work by the author [3] applied fitness sharing to grammar-guided genetic programming, and demonstrated dramatic reductions in error at convergence on three problems: the 6- and 11-multiplexer problems, and learning recursive list membership. A second paper [4] introduced populations of partial functions, and demonstrated a small but significant improvement in performance of the fittest population member at convergence on recursive list membership, but inconclusive results on the two multiplexer problems. A difficulty in that work lay in comparing partial and total individuals, since a partial function by definition does not cover the whole of the learning set, so in some problems it was necessary to gradually introduce evolutionary pressure toward totality in order to ensure comparability. It was conjectured there that populations of partial functions might be better suited to committee learning approaches, since these remove the necessity for the evaluated individuals to cover all learning cases. This paper investigates that conjecture, and presents further results. For completeness, the two fitness sharing methods are also compared with the use of raw fitness in populations of partial and total functions. It had previously been noted that generations using partial functions appeared to complete faster than those using total functions; timings for the runs in these experiments were kept, and a comparison made.

2. Details of Method

2.1. Grammar Guided Genetic Programming

This work makes use of the grammar-guided genetic programming paradigm [5] in the form of the DCTG-GP programming system [6]. In grammar-guided GP, the search space is represented by a context free grammar, and the individuals are represented by parse trees in that grammar. The function represented by each individual is determined by the leaves of the parse tree (in standard terminology these are called terminals, but that gives rise to a terminological conflict with the language of genetic programming, because a function symbol is a leaf of the parse tree, but would be regarded as a nonterminal in established GP terminology). DCTG-GP uses Definite Clause Transformation [7] to define both the syntax and the semantics of the object programs, so that the semantics are closely tied to the syntax. However for the work described here, the grammars were used simply to define a strongly-typed syntax, so that the work would extend to strongly typed genetic programming [8] extended with a suitable mechanism for representing an undefined value.

2.2. Implicit Fitness Sharing

Implicit fitness sharing makes the assumption that the fitness of each individual in the population is calculated as the sum of rewards for a number of cases:

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

The implicitly shared fitness is then calculated by dividing the reward amongst all individuals which make the same decision for that case:

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

2.3. Partial Functions

By a partial function we mean a function whose value, for some arguments, is undefined. In this work, this is achieved by adding, for each nonterminal in the grammar, a production leading to a distinguished 'undefined' value. For example, the following might be a fragment of a grammar for boolean functions:

BOOL $\rightarrow$ BOOL $\wedge$ BOOL
BOOL $\rightarrow$ BOOL $\vee$ BOOL
BOOL $\rightarrow$ $\neg$ BOOL
BOOL $\rightarrow$ UNDEF

Semantically, UNDEF always evaluates to the undefined value ϕ. In this work, lazy evaluation is used:

$$\text{true} \wedge \phi = \phi \wedge \text{true} = \phi$$

$$\text{false} \wedge \phi = \phi \wedge \text{false} = \text{false}$$

but eager evaluation would not be inconsistent with the approach.

2.4. Combining Partial Functions and Fitness Evaluation

When functions are permitted to be partial rather than total, we can distinguish between the accuracy of an individual and its coverage - an individual may be highly accurate on those cases which it chooses to predict, but that may be a small percentage of the overall set of cases, so the individual may have poor coverage. This is the core idea behind the introduction of partial functions to genetic programming, permitting individuals to concentrate on those parts of the problem on which they perform best.

Hence it is appropriate to reward individuals for accuracy rather than coverage, and rely on the overall population-based evaluation strategy to ensure coverage. Since implicit fitness sharing will place evolutionary pressure on the overall population to diversify, we can expect it to provide complete coverage by the population as a whole. Hence in the fitness sharing runs, the fitness evaluation used accuracy as the fitness measure (ie the overall fitness is the shared fitness divided by the number of cases for which the individual is defined).

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

Where partial functions are evaluated using raw fitness, if accuracy were used as the measure, there would be no pressure on the population to provide complete coverage of the cases. Hence for these runs, coverage is used as the measure of overall fitness - an undefined value is simply treated as an incorrect answer.

2.5. Committee Evaluation

There are many possible ways of evaluating the predictions of a population. In this work, a simple weighted voting approach was used. Preliminary experiments with linearly-scaled voting suggested that this gave insufficient weight to the fittest members of the population, so the voting weights used the square of the fitness of the individual. The prediction of the population as a whole on a given case was taken to be the prediction with the highest voting weight (for partial functions, 'undefined' predictions were not counted).

3. Experimental Design

Three experimental problems were used:

- the 6-multiplexer problem
- the 11-multiplexer problem
- recursive list membership

The 6-multiplexer problem is to predict, from the inputs, the outputs of a multiplexer having two address and four data lines. The search space is the set of boolean combinations of the address and data values using 'and', 'or', 'not' and three-way 'if' combinators.

Table 1: 6-multiplexer Grammar

EXPR $\rightarrow$ BOOL
BOOL $\rightarrow$ TERM
BOOL $\rightarrow$ and BOOL BOOL
BOOL $\rightarrow$ or BOOL BOOL
BOOL $\rightarrow$ not BOOL
BOOL $\rightarrow$ if BOOL BOOL BOOL
TERM $\rightarrow$ a0
TERM $\rightarrow$ a1
TERM $\rightarrow$ d0
TERM $\rightarrow$ d1
TERM $\rightarrow$ d2
TERM $\rightarrow$ d3

The 11-multiplexer problem is to predict the outputs of a three address, eight data line multiplexer; the grammar is extended with terminals for the additional inputs.

The recursive list membership problem is to learn, from a set of positive and negative cases of list membership, an expression for a recursive membership function using a lisp-like language.

Table 2: List Membership Grammar

S → M
M → if EXPN EXPN M
M → ''
EXPN → atom LST
EXPN → eq LST LST
EXPN → member x LST
EXPN → true
EXPN → false
LST → first LST
LST → rest LST
LST → x
LST → y

When a system is to learn recursive functions, an additional choice has to be made: whether the recursions are to be evaluated intensionally (ie from further calls to the defined function) or extensionally (ie from the actual values in the data). There are problems with extensional evaluation with incomplete datasets, so we chose to use intensional evaluation. However this introduces an additional problem, namely the risk that a recursion may be infinite. Rather than attempt to detect infinite loops syntactically (in general, an uncomputable problem), we have chosen a simple semantic expedient: a count is kept of the depth of recursion, and any depth greater than a fixed count evaluates as 'loop'. The 'loop' value is treated much like the 'undefined' value, except that at the top level it is treated as an 'incorrect' answer (other choices could be made here, but they would lead to difficulties in comparing partial and total function populations). The fixed depth was chosen as 15, on the basis that the longest lists presented as cases have 10 elements, so further recursion is unlikely to be productive.

The experiments used half-ramped initialisation and tournament selection. Parameter settings are given below: They differ from the experiments reported in earlier papers [3,4] chiefly in running smaller populations for more generations, as some difficulty had been experienced in that work in ensuring that the runs actually reached convergence. Longer runs and smaller populations would both be expected to result in earlier convergence, and thus permit differences to be seen within the computationally accessible region.

Table 3: Multiplexer GP Parameters

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

For the 6 multiplexer, the raw fitness was the proportion of the 64 cases correctly predicted. For the 11 multiplexer, computational considerations prevented use of all 2048 cases. Instead, each generation a random selection of 64 cases was made, and evaluation conducted against those. An independent selection was made at each generation.

The 6-multiplexer runs were terminated at 200 generations, and the 11-multiplexer at 500 generations, or earlier if population evaluation gave a 100% accurate solution. In the 11-multiplexer case, the population was tested only against the 64 cases in the final generation. This might have resulted in the acceptance of populations which would not extend accurately to the full 2048 case set, but this does not affect the comparative evaluations which are the focus of this work.

Table 4: List Membership GP Parameters

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

For the list membership problem, the population was evaluated against 20 cases, 10 positive and 10 negative.

Table 5: List Membership Cases

POSITIVE CASES	NEGATIVE 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])

Each experiment involved four separate treatments:
- raw fitness, populations of total functions
- fitness sharing, populations of total functions
- raw fitness, populations of partial functions
- fitness sharing, populations of partial functions

4. Results

4.1. 6-Multiplexer Results

The percentage of runs which terminated in a correct voted solution are shown in table 6 and plotted in figure 1:

Table 6: Percentage of Runs Generating Correct Solution

	Total Functions	Partial Functions
Raw Fitness	61	4
Fitness Sharing	98	98

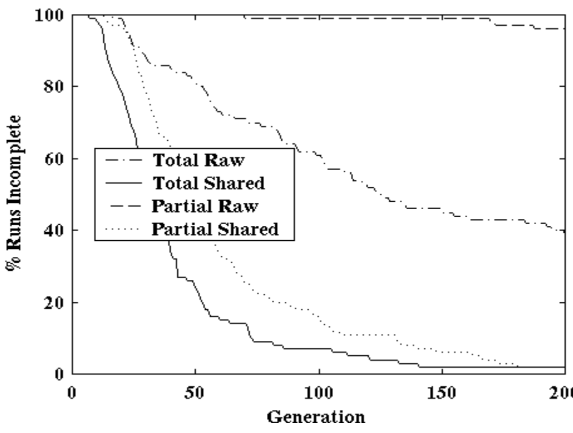

Figure 1: % Runs Incomplete, 6 Multiplexer

Figure 2 shows the mean of the depth of the trees used for the six multiplexer problem (generations before 30, and the partial functions with raw fitness run, are omitted from the plot in order to concentrate on the area of interest). The trees in the partial functions / shared fitness treatment are in general considerably smaller up to about generation 150, at which point the 'total shared' and 'partial shared' plots are based only on the two remaining incomplete runs, and so may not be reliable indicators.

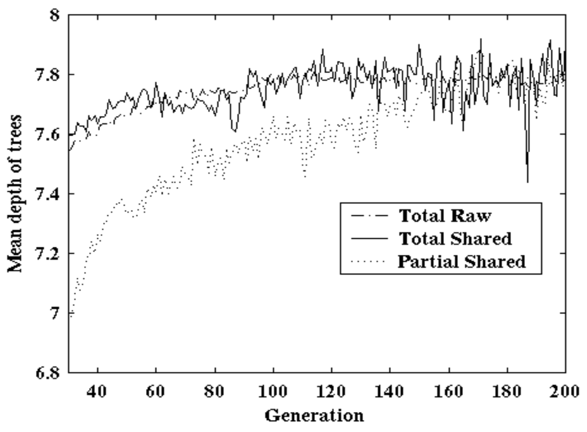

Figure 2: Mean Depth of Trees, 6 Multiplexer

Table 7: Computational Time, 6 Multiplexer

	Raw Fitness		Fitness Sharing	
	Total	Partial	Total	Partial
cpu/Run	1464.01	1038.97	373.49	410.79
Generations/Run	128.06	198.00	41.94	61.31
cpu/Generation	11.6	5.27	8.68	6.47

cpu times are in seconds

4.2. 11-Multiplexer Results

The results for the 11 multiplexer are shown in tables 8 and 9 and plotted in figures 3 and 4:

Table 8: Percentage of Runs Generating Correct Solution

	Total Functions	Partial Functions
Raw Fitness	11	0
Fitness Sharing	81	72

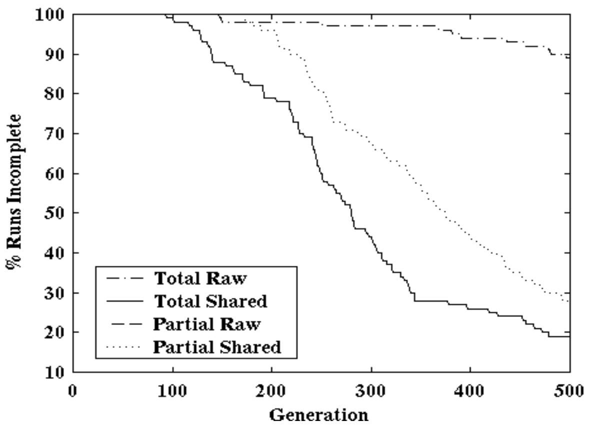

Figure 3: % Runs Incomplete, 11 Multiplexer

Tree depth results and run-times for the 11 multiplexer are shown in table 9 and figure 4 (early generations and 'partial raw' run are omitted to concentrate on the area of interest).

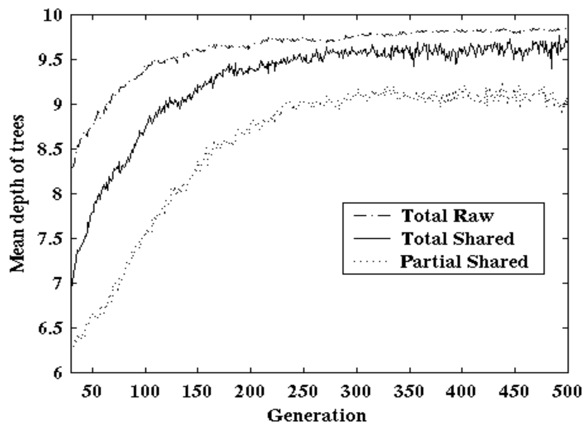

Figure 4: Mean Tee Depth, 11 Multiplexer

Table 9: Computational Time, 11 Multiplexer

	Raw Fitness		Fitness Sharing	
	Total	Partial	Total	Partial
cpu/Run	8257.80	3109.31	4371.97	3823.90
Generations/Run	485.34	500.00	305.04	371.21
cpu/Generation	16.99	6.22	13.93	10.23

cpu times are in seconds

4.3. List Membership Results

The list membership results are shown in tables 10 and 11, and figures 5 and 6.

Table 10: Percentage of Runs Generating Correct Solution

	Total Functions	Partial Functions
Raw Fitness	42	32
Fitness Sharing	62	57

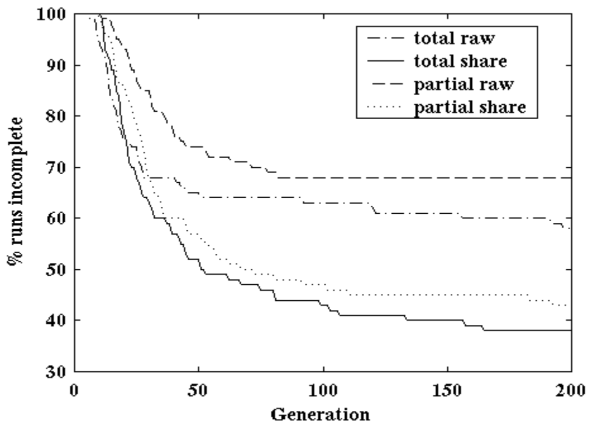

Figure 5: % Runs Incomplete, List Membership

Table 11: Computational Time, List Membership

	Raw Fitness		Fitness Sharing	
	Total	Partial	Total	Partial
cpu/Run	2669.03	2841.90	1600.10	1597.01
Generations/Run	131.89	147.13	100.19	109.22
cpu/Generation	20.53	18.59	15.20	14.61

cpu times are in seconds

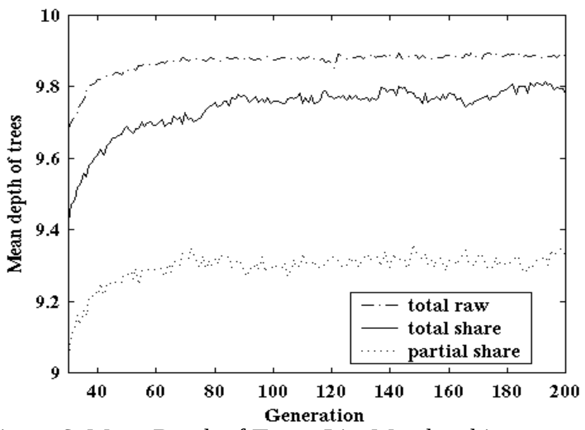

Figure 6: Mean Depth of Trees, List Membership

5. Discussion

The results are quite clear on one point: implicit fitness sharing outperforms raw fitness by large margins in every experiment. This reinforces the results of [3], but demonstrates that the differences are even larger with committee learning than with individual evaluation. In general, the results on comparison of partial and total functions are more equivocal. It is unsurprising that partial functions perform poorly when combined with raw fitness; the experiments were included for completeness, rather than because the approach made sense.

On the issue which was the main focus of this work, the comparison of partial and total functions using fitness sharing, there is no strong result: the behaviour at convergence of the two treatments is very similar. This could be seen as an implicit argument in favour of populations of total functions: if the convergent behaviour of both is similar, but the greater eagerness of the total function search results in lower early fitness, then total functions might be preferred. However the timing results give the lie to this: the partial function timings are very similar to those for total functions, even though the partial function populations evaluated considerably more generations overall. Presumably this arises from reduced complexity of the partial functions, as evidenced also by the average depth plots. Thus the greater eagerness of the total function search is offset by the faster per-generation evaluation of the partial function populations, and there is little to choose between the two in computation time.

This suggests a further issue. The work carried out here has looked simply at performance on noise-free and complete (or in the case of the 11-multiplexer problem, near-complete) datasets, hence generalisation is not an issue. But the primary aim of committee learning approaches in particular is to improve generalisation ability. The smaller depth and faster evaluation of the partial function populations suggest that they have been more subject to Occam's razor; in turn, this would suggest that they are less likely to be overfitted to the training data, and may generalise better. The next phase of this work will investigate the relative generalisation ability of partial and total function populations on both synthetic and real-world datasets.

6. Conclusions

The results presented here confirm that committee learning is well suited to the use of fitness sharing, and far better results are obtained than with raw fitness. The results are highly equivocal on the comparison between populations of partial and of total functions, and do not give strong grounds for preferring either over the other.

Acknowledgments

The ideas in this paper have benefited greatly from discussions over the years with Paul Darwen, Peter Whigham, Xin Yao, Ko-Hsin Liang, Hussein Abbass, and other members of the Machine Intelligence and Communication Group at the Australian Defence Force Academy. 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. My thanks also to a number of anonymous reviewers of this research for their contributions to details of both the experimental work and its exposition.

References

[1] 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

[2] 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

[3] McKay, R I: 'Fitness Sharing in Genetic Programming', *Proceedings, GECCO 2000*

[4] McKay, R I 'Partial Functions in Fitness-Shared Genetic Programming', *Proceedings, CEC 2000*

[5] 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

[6] Ross, B J: 'Logic-based Genetic Programming with Definite Clause Translation Grammars', *Proceedings GECCO-99*, Morgan Kaufmann, 1999.

[7] Abramson, H and Dahl, V *'Logic Grammars'*, Springer-Verlag, 1989

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