Promoting Phenotypic Diversity in Genetic Programming

David Jackson

Dept. of Computer Science, University of Liverpool
Liverpool L69 3BX, United Kingdom
djackson@liverpool.ac.uk

Abstract. Population diversity is generally seen as playing a crucial role in the ability of evolutionary computation techniques to discover solutions. In genetic programming, diversity metrics are usually based on structural properties of individual program trees, but are also sometimes based on the spread of fitness values in the population. We explore the use of a further interpretation of diversity, in which differences are measured in terms of the behaviour of programs when executed. Although earlier work has shown that improving behavioural diversity in initial GP populations can have a marked beneficial effect on performance, further analysis reveals that lack of behavioural diversity is a problem throughout whole runs, even when other diversity levels are high. To address this, we enhance phenotypic diversity via modifications to the crossover operator, and show that this can lead to additional performance improvements.

1 Introduction

It is generally accepted that a problem associated with genetic programming and other forms of evolutionary computation is that population diversity tends to diminish over time [1, 2], and that this may then lead to convergence on local minima from which the evolutionary process cannot escape to explore the search landscape more widely for solutions. However, more rigorous probing into the whole issue of diversity and its possible impact reveals that there is little consensus as to how it should be defined, measured, analysed or promoted.

Broadly speaking, diversity in genetic programming refers to how different each individual is from other members of the host population, and although this notion of 'difference' may be interpreted in a variety of ways, previous work has tended to categorise diversity as falling into two camps. The first is *genotypic*, or structural, diversity, which is based upon differences between the structure of the trees or other representations used to encode individual programs. The second category is *phenotypic* diversity. Ostensibly, this is defined in terms of the behaviour rather than the structure of programs, but in practice it has usually corresponded to the spread of fitness values amongst members of the population.

We shall say more about these previously established views of diversity in the next section on related work. Following that, in Section 3 we will describe an alternative

R. Schaefer et al. (Eds.): PPSN XI, Part II, LNCS 6239, pp. 472–481, 2010.

view of phenotypic diversity which is based not on fitness values, but on the behaviour of individuals when they are executed. In Section 4 we will compare the extent of the various forms of diversity in several problem domains, with particular emphasis on how it changes during the lifetime of GP runs. In Section 5 we build on earlier work concerned with creating more diverse initial populations [3] by describing methods for maintaining that diversity in subsequent generations, then go on to describe and compare the effects of these techniques in experimentation. Finally, Section 6 draws some conclusions and gives pointers to further work.

2 Related Work

As it relates to genetic programming, the term 'diversity' has a variety of interpretations, and hence a number of different ways have been proposed for measuring it, creating it and maintaining it. Overviews of diversity measures can be found in [4] and [5], while Burke et al [6] give a more extensive analysis of these measures and of how they relate to fitness.

The most common usage of the term is concerned with differences in the *structure* of individual program trees; that is, in their size, their shape, and in the functions and terminals used at individual nodes. Recognizing the importance of including a wide range of structures in the initial population, Koza [7] proposed the use of a 'ramped half-and-half' algorithm, and many implementations have continued to follow his advice. The approach is claimed to give good diversity in the structure of program fragments which can then be combined and integrated to produce more complex and hopefully fitter programs.

Measurements of structural diversity may involve nothing more than simple node-for-node comparison of program trees;. More sophisticated structural diversity metrics may be based on edit distance [8], where the similarity between two individuals is measured in terms of the number of edit operations required to turn one into the other.

A difficulty with comparing individuals based on their apparent structure is that program trees which are seemingly very different in appearance may in fact compute identical functions. Seeing beyond these surface differences requires the use of graph isomorphism techniques, but these are computationally expensive and become even more so as program trees grow larger over time. A simpler, less costly alternative is to check for pseudo-isomorphism [5], in which the possibility of true isomorphism is assessed based on characteristics such as tree depth and the numbers of terminals and functions present. However, the accuracy of this assessment may be subject to the presence of introns in the code; Wyns et al [9] describe an attempt to improve on this situation through the use of program simplification techniques to remove redundant code.

In contrast, *behavioural* or *phenotypic* diversity metrics are based on the functionality of individuals, i.e. the execution of program trees rather than their appearance. Usually, behavioural diversity is viewed in terms of the spread of fitness values obtained on evaluating each member of the population [10]. One way of measuring such diversity is by considering the fitness distribution as an indicator of

entropy, or disorder, in the population [11, 9]. Other approaches consider sets or lists of fitness values and use them in combination with genotypic measures [12, 13]. For certain types of problem it may be possible to achieve the effect of behavioural diversity without invoking the fitness function, via the use of semantic sampling schemes [14]. Semantic analysis of programs is also used in the diversity enhancing techniques described by Beadle and Johnson [15, 16].

3 Phenotypic Diversity

Our own approach to diversity differs from others in that it does not involve structural considerations, fitness values or semantic analysis of programs. Instead, it focuses on the observed behaviour of individuals when they are executed. To investigate this fully, we have applied it to a variety of problem domains; these comprise two Boolean problems (6-multiplexer and even-4 parity), two navigation problems (Santa Fe and maze traversal), and one numeric problem (symbolic regression of a polynomial).

The 6-mux, even-4 parity and Santa Fe problems are all standard benchmarks in GP, and further details can be found elsewhere (e.g. Koza [7]). In our symbolic regression problem we attempt to evolve a program equivalent to the polynomial $4x^4 - 3x^3 + 2x^2 - x$. The fitness cases consist of 32 x-values in the range [0,1), starting at 0.0 and increasing in steps of 1/32, plus the corresponding y-values. Other than this, the problem is again fairly standard. Our second navigation problem is that of finding a route through a maze. Although less well-known than the ant problem, it has been used as the subject for research on introns in several studies [17-19], and again details can be found in those papers and in our earlier paper on this topic [3]. Other parameters as they apply to the experiments described in the remainder of this paper are shown in Table 1.

Table 1. GP system parameters common to all experiments

Population size	500
Initialisation method	Ramped half-and-half
Evolutionary process	Steady state
Selection	5-candidate tournament
No. generations	51 generational equivalents (initial+50)
No. runs	100
Prob. crossover	0.9
Mutation	None
Prob. internal node used as crossover pt.	0.9

In the case of the Boolean problems, the behaviour of an individual is measured in terms of the outputs it produces; this is recorded as a string of binary values for each of the test cases used during fitness evaluation. So, for the 6-mux problem, there is a 64-bit string associated with each member of the population, while for the even-4 parity problem only a 16-bit string need be recorded. To save memory, these can be packed into 64-bit and 16-bit integers, respectively. We say that two individuals

exhibit phenotypic differences if they differ in any of the corresponding bits in their output strings, and that an individual is phenotypically unique in a population if there are no other members of that population with exactly the same binary output string.

For the symbolic regression problem, we again record the outputs produced for each test case, but this time it is a vector of 32 floating point results. In comparing phenotypes we choose not to look for exact matches, but instead check whether the differences between corresponding outputs lie within some pre-defined value epsilon. Hence, two individuals are said to be behaviourally identical if, for each x-value, the absolute difference between the corresponding y-values is less than epsilon. The value for epsilon was arbitrarily chosen to be the same as that used to check for 'hits' in fitness assessment, i.e. 0.01.

For the two navigation problems, the situation is complicated by the fact that the evolving programs do not produce outputs as such: their behaviour is measured in terms of the movements produced by function and terminal execution. Because of this, the record we make of an individual's behaviour is the sequence of moves it makes in the grid or maze during execution. We are not concerned with recording any left or right turns that are executed while remaining on a square, nor with any decision making via execution of statements such as IF_FOOD_AHEAD or WALL_AHEAD.

To record the path histories, we associate with each individual in the population a vector of {north, east, south, west} moves. Each time the executing program moves to a different square, the heading is added to the end of the vector. Since a program times-out after a fixed number of steps (600 for the ant problem, 1000 for the maze), we know that the vector cannot be longer than that number of elements per individual, and so memory occupancy is not a huge issue. Determining behavioural differences between individuals becomes simply a matter of comparing these direction vectors.

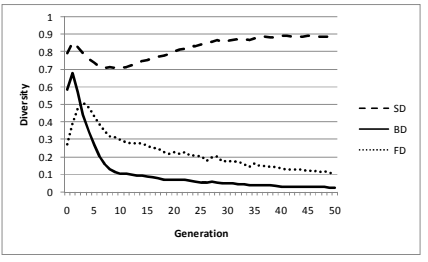
4 Measuring Diversity

The first thing we wish to explore is the extent to which diversity is present in populations, and how it alters over the lifetime of each run. For this, we need to define appropriate metrics.

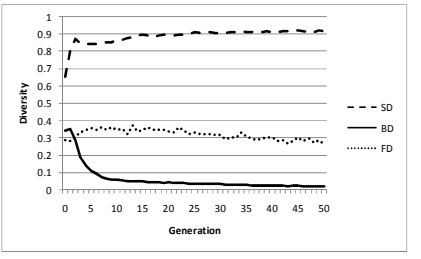
We start with structural diversity (SD). This is found by counting the number of distinct program structures present in the population and then dividing this by the population size. Thus, SD = 1 if and only if every member of the population has a different structure, whereas a value of SD close to zero indicates poor structural diversity (i.e. much duplication). Behavioural diversity (BD) is defined in a very similar way, by counting up the number of distinct behaviours and again dividing by the size of the population. As before, BD = 1 only when each individual's behaviour is unique.

Fitness diversity (FD) is calculated in a slightly different manner. Unlike either structural or behavioural diversity, in which every member of the population can be unique, the range of possible fitness values tends to be much smaller than the population size, and varies from problem to problem. We therefore define fitness diversity (FD) to be the number of distinct fitness values found in the population divided by the number of possible values. For example, if a given population in the

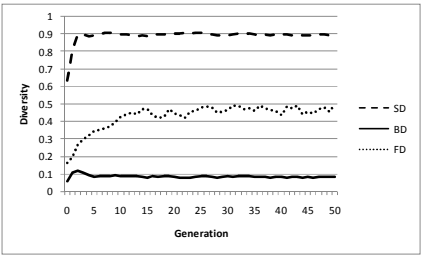
even-4 parity problem contained a total of 5 different fitness values, then its FD would be 5/17 = 0.294, since the problem allows for 17 possible fitness values (0-16). On this scale, a value of 1.0 would indicate full fitness diversity (which would also imply that the population contained a solution!).



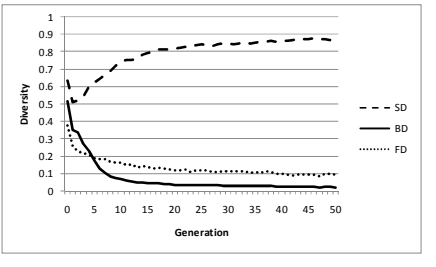
(a) Ant



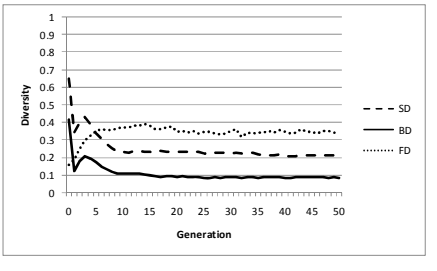
(b) Parity



(c) Maze



(d) Mux



(e) Regression

Fig. 1. Diversity changes for each test problem, averaged over 100 runs

Given these definitions, we can plot the changes that take place in each form of diversity. The graphs of Figure 1 show this for each of our benchmark problems. The figures are averaged over 100 runs. The first observation that can be made about these graphs is that, in four of the five problems, structural diversity increases rather than decreases, and tends to reach quite a high level of about 0.9. In other words, only

about 10% of individuals have structural clones elsewhere in the population. This suggests that crossover is successful at generating structurally unique program trees, and that efforts to increase structural diversity in these problems may not have a substantial impact. Only in the case of the symbolic regression problem does structural diversity fall significantly, dropping to just over 0.2 from generation 10 onwards.

The picture is more mixed for fitness diversity: in the ant problem it rises and then falls to below its initial level; in the maze and regression problems it rises steadily and then remains fairly constant; in the parity problem it tends to hover around a value of 0.3; and in the multiplexer problem it falls from 0.4 down to about 0.1.

In all of the problems it is behavioural diversity that comes off worst. Whatever its initial value (which in some cases can be comparatively good), it always drops to below 0.1. In the even parity and mux problems it reaches as low as 0.02 in generation 50; this corresponds to only 10 distinct behaviours in a population of 500. For most of the problems there is a wide gap between the SD and BD graphs, indicating that many programs that are structurally dissimilar do not differ at all in their behaviour. The number of distinct behaviours in the parity and mux problems dip to only 10 despite the number of distinct structures in the population climbing to about 450.

5 Preserving Diversity

The experiments of the previous section show that there is relatively little opportunity to increase the structural diversity of a population, but much greater scope for increasing behavioural diversity. Fitness diversity also remains low, but doing something about this is far more problematic. In initial GP populations, most members will have poor fitness, and so the number of distinct fitness values can be expected to be quite small. Increasing diversity in these early stages of evolution would involve introducing some fitter individuals into the population, but this begs the question of how one goes about finding these fitter programs without resorting to the evolutionary process that subsequent generations are meant to implement!

Our approach to promoting diversity is a two-step process. The first phase is to establish increased diversity in the initial population. This can be done by making changes to the ramped half-and-half algorithm mentioned earlier so that duplicates (either structural or behavioural) are eliminated. This approach was covered in detail in an earlier paper [3].

The next task is to attempt to preserve diversity throughout the remainder of the evolutionary process. There will be pressure to reduce diversity because of the reproduction operator, which simply clones relatively fit individuals. This will be counterbalanced to an extent by the recombination operator, which creates new individuals via subtree crossover. In most problems, the nature of the terminal and function sets, and the way in which crossover points are chosen, is sufficient to ensure that entirely novel structures are created. This explains the continually high SD levels in the earlier graphs (the regression problem being an exception).

Unfortunately, crossover does not always introduce new behaviours. A primary reason for this is that crossover often takes place at the site of introns, particularly during the latter stages of evolution in which bloat is extensive. These introns are

often non-executed pieces of code [19], and so the crossover operation creates a child which does not differ behaviourally from the parent receiving the subtree. It is for these reasons that we have chosen the crossover operator as the focal point for changes to maintain at least some of the diversity already introduced during initialisation. The changes are simple, as the following pseudo-code shows:

```
function crossover
   childnum = 0
   select parents by tournament
   select member to be replaced
   do
      establish crossover points
      create child
      childnum = childnum + 1
   while (child same as a parent
         and childnum < MAX_BROOD)
endfunction
```

The parents and the individual to be replaced are selected by tournament. The code then enters a loop, choosing crossover points and generating offspring until a child is found which differs from both its parents. The comparison between a child and its parents can be either structural or behavioural. In both cases the variable *childnum* is used to prevent an excessive number of offspring from being produced. We used a value of 20 for MAX_BROOD, but found in practice that the actual number of children generated in each crossover usually fell far short of that, even when searching for behavioural differences.

Table 2. Effects of diversity promotion on solution discovery rate, given as number of solutions found in 100 runs

Problem	Standard GP	SD-Initial	SD-All	BD-Initial	BD-All
6-mux	56	66	70	79	99
Even-4	14	11	11	23	54
Regress	10	10	18	24	36
Ant	13	9	12	18	25
Maze	14	18	17	51	95

In Table 2 we show the effects of these algorithms on the rate of success at finding solutions to our benchmark problems. For each problem, we first of all give the success rate of our standard GP system, measured as the number of solutions found in 100 runs. We also wish to distinguish between the effects of using our first algorithm in isolation (i.e. eliminating duplicates in the initial population only), and using both algorithms together to promote diversity throughout the lifetime of each run. Hence, the column labelled SD-Initial shows what happens when structural clones are eliminated in the initial population only, while the column labelled SD-All shows the effects of augmenting this initialisation phase with the subsequent attempts to prevent structural duplicates being created during crossover. The columns headed BD-Initial

and BD-All should be interpreted in the same manner, but based on the promotion of behavioural rather than structural diversity.

What we can see from Table 2 is that the removal of structural duplicates does not always have a beneficial effect on solution finding performance. It is worth noting, however, that the biggest positive impact of the combined approach to structural diversity is on the symbolic regression problem, where the success rate is almost doubled. Interestingly, it was the regression problem which the graphs of Fig. 1 revealed to have the most scope for improvement in the structural diversity levels.

The situation with regard to behavioural diversity appears more promising. Even if we restrict our diversity enhancing measures to the initial population, a substantial increase in performance follows for most of our problems, particularly so for maze traversal, where the success rate jumps from 14% to 51%. Once we add in the second algorithm to preserve behavioural diversity throughout the remainder of each run, performance improves even further, with near perfect records being seen for the maze and 6-mux problems. A t-test ($p < 0.05$) performed on the best fitness values found at the end of each run indicates that the improvements are statistically significant.

To make the comparison fair, we have to ask at what cost our improvements are obtained. One commonly used method of comparing cost is Koza's computational effort metric [7]. However, this assumes a fixed number of fitness evaluations per generation, which for our purposes is not applicable because of the additional effort required both to create the initial populations and to assess each member of the broods created in our amended crossover operator.

The cost metric we shall use instead is a count of the number of fitness evaluations performed over all 100 runs, divided by the number of solutions found. This gives us a measure of effort in terms of the number of evaluations per solution. Table 3 gives these figures for each of our problems and diversity enhancing mechanisms.

Table 3. Effects of diversity promotion on computational effort, measured as number of fitness evaluations per solution

Problem	Standard GP	SD-Initial	SD-All	BD-Initial	BD-All
6-mux	23263	16416	18261	12140	7703
Even-4	151518	195781	208667	101570	58662
Regress	217612	217486	142659	88430	71465
Ant	158498	240285	193817	118068	119118
Maze	150959	115998	130956	8329286	4482422

In most cases, it would seem that when it comes to counting the computational cost of finding solutions, our approaches to behavioural diversity enhancement offer significant improvements over standard GP, and over systems that attempt to increase structural diversity. There is, however, a notable exception. In the maze problem, the effort involved in creating the initial behaviourally diverse population is immense; and although a dramatic increase in the success rate is observed, this is still not enough to bring the number of evaluations per solution down to a level that is competitive with standard GP.

6 Conclusions

It is accepted wisdom that lack of diversity in evolutionary computing techniques such as genetic programming can hamper the search for solutions, and that diversity tends to diminish over the lifetime of a run. The inference to be drawn from this is that efforts to promote and maintain diversity should have a beneficial effect. Our research supports this, but with the caveat that it very much depends on how the notion of diversity is interpreted.

We have distinguished in this paper between genotypic and phenotypic diversity in GP. The former is associated with the structure of program trees making up the population; the latter is usually defined as corresponding to the spread of fitness values in a population. We have introduced a further interpretation of phenotypic diversity, defined in terms of the recorded behaviour of programs as they execute. We have also shown that, despite great differences in the nature of problems for which GP may be used, the approach is a general one applicable to a wide range of domains.

When assessing the extent of the various forms of diversity experimentally we find that, far from diminishing, structural diversity tends to increase early on in each run, and then remain comparatively high throughout the remainder of the run. By contrast, behavioural diversity (in the form we have described it) does fall and remain at very low levels. Accordingly, when we introduce algorithms to promote structural diversity, we see little in the way of performance gains. The greatest improvement is seen in the symbolic regression problem; it is perhaps no coincidence that it is this problem which has the biggest scope for increasing its structural diversity levels.

Again in contrast, when the algorithms are used to create and preserve behavioural diversity, we see a substantial improvement in the solution finding performance for all the problems we studied. Although additional fitness evaluations are required to generate an initially diverse population, and then to assess broods of children to preserve that diversity during crossover, this is outweighed by the improved success rate, so that the computational costs per solution are significantly reduced. The one exception to this in our problem set is maze traversal. The reason for this is that the physical constraints of the maze make it difficult to generate a set of unique behaviours when using an unintelligent initialization algorithm.

One way of combating this problem would be to place an upper limit on the number of new programs that are created for each member that enters the population. This would enhance diversity but allow some duplication. Indeed, there is a general research issue here as to how different levels of diversity may affect performance, and we hope to explore this further. More generally, we plan to investigate the potential for exploiting phenotypic diversity in the form we have described it.

References

1. McPhee, N.F., Hopper, N.J.: Analysis of Genetic Diversity through Program History. In: Banzhaf, W., et al. (eds.) Proc. Genetic and Evolutionary Computation Conf., Florida, USA, pp. 1112–1120 (1999)
2. Daida, J.M., Ward, D.J., Hilss, A.M., Long, S.L., Hodges, M.R., Kriesel, J.T.: Visualizing the Loss of Diversity in Genetic Programming. In: Proc. IEEE Congress on Evolutionary Computation, Portland, Oregon, USA, pp. 1225–1232 (2004)

3. Jackson, D.: Phenotypic Diversity in Initial Genetic Programming Populations. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 98–109. Springer, Heidelberg (2010)
4. Hien, N.T., Hoai, N.X.: A Brief Overview of Population Diversity Measures in Genetic Programming. In: Pham, T.L., et al. (eds.) Proc. 3rd Asian-Pacific Workshop on Genetic Programming, Hanoi, Vietnam, pp. 128–139 (2006)
5. Burke, E., Gustafson, S., Kendall, G., Krasnogor, N.: Advanced Population Diversity Measures in Genetic Programming. In: Guervos, J.J.M., et al. (eds.) PPSN 2002. LNCS, vol. 2439, pp. 341–350. Springer, Heidelberg (2002)
6. Burke, E., Gustafson, S., Kendall, G.: Diversity in Genetic Programming: An Analysis of Measures and Correlation with Fitness. IEEE Transactions on Evolutionary Computation 8(1), 47–62 (2004)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
8. de Jong, E.D., Watson, R.A., Pollack, J.B.: Reducing Bloat and Promoting Diversity using Multi-Objective Methods. In: Spector, L., et al. (eds.) Proc. Genetic Evolutionary Computation Conf., San Francisco, CA, USA, pp. 11–18 (2001)
9. Wyns, B., de Bruyne, P., Boullart, L.: Characterizing Diversity in Genetic Programming. In: Collet, P., et al. (eds.) EuroGP 2006. LNCS, vol. 3905, pp. 250–259. Springer, Heidelberg (2006)
10. Rosca, J.P.: Genetic Programming Exploratory Power and the Discovery of Functions. In: McDonnell, J.R., et al. (eds.) Proc. 4th Conf., Evolutionary Programming, San Diego, CA, USA, pp. 719–736 (1995)
11. Rosca, J.P.: Entropy-Driven Adaptive Representation. In: Rosca, J.P. (ed.) Proc. Workshop on Genetic Programming: From Theory to Real-World Applications, Tahoe City, CA, USA, pp. 23–32 (1995)
12. D'haeseleer, P., Bluming, J.: Effects of Locality in Individual and Population Evolution. In: Kinnear, K.E., et al. (eds.) Advances in Genetic Programming, ch. 8. pp. 177–198. MIT Press, Cambridge (1994)
13. Ryan, C.: Pygmies and Civil Servants. In: Kinnear, K.E., et al. (eds.) Advances in Genetic Programming, ch.11, pp. 243–263. MIT Press, Cambridge (1994)
14. Looks, M.: On the Behavioural Diversity of Random Programs. In: Thierens, D., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf. (GECCO 2007), London, England, UK, pp. 1636–1642 (2007)
15. Beadle, L., Johnson, C.G.: Semantic Analysis of Program Initialisation in Genetic Programming. Genetic Programming and Evolvable Machines 10(3), 307–337 (2009)
16. Beadle, L., Johnson, C.G.: Semantically Driven Crossover in Genetic Programming. In: Proc. IEEE Congress on Evolutionary Computation (CEC), Hong Kong, pp. 111–116 (2008)
17. Soule, T.: Code Growth in Genetic Programming. PhD Thesis, University of Idaho (1998)
18. Langdon, W.B., Soule, T., Poli, R., Foster, J.A.: The Evolution of Size and Shape. In: Spector, L., et al. (eds.) Advances in Genetic Programming, vol. 3, pp. 163–190. MIT Press, Cambridge (1999)
19. Jackson, D.: Dormant Program Nodes and the Efficiency of Genetic Programming. In: Beyer, H.-G., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf. (GECCO 2005), Washington DC, USA, pp. 1745–1751 (2005)