Matrix Analysis of Genetic Programming Mutation

Andrew J. Parkes, Ender Özcan, and Matthew R. Hyde

School of Computer Science
The University of Nottingham
Nottingham, NG8 1BB
United Kingdom (UK)
E-mail: [ajp,exo,mvh]@cs.nott.ac.uk
URL: http://cs.nott.ac.uk/∼[ajp,exo,mvh]

Abstract. Heuristic policies for combinatorial optimisation problems can be found by using Genetic programming (GP) to evolve a mathematical function over variables given by the current state of the problem, and whose value is used to determine action choices (such as preferred assignments or branches). If all variables have finite discrete domains, then the expressions can be converted to an equivalent lookup table or 'decision matrix'. Spaces of such matrices often have natural distance metrics (after conversion to a standard form). As a case study, and to support the understanding of GP as a meta-heuristic, we extend previous bin-packing work and compare the distances between matrices from before and after a GP-driven mutation. We find that GP mutations often correspond to large moves within the space of decision matrices. This gives evidence that the role of mutations within GP might be somewhat different than their role within Genetic Algorithms.

1 Introduction

The effects of the genetic programming (GP) mutation operator are not often analysed in detail. When an analysis is performed, it is often to show how successful different levels of mutation are, or to show the effects of its interaction with crossover. This paper presents a methodology to analyse the effect of mutation on the phenotype of an individual. Mutation's effects on the genotype are trivial to calculate, as it is easy to see how the new and old expressions differ. The effects are less easy to quantify on the phenotype, yet this is often the effect which is most important.

In GP, the genotype and phenotype are often indistinguishable, but there are many applications of GP where they are clearly different. One example is when GP is used to generate heuristic functions, which give a score to a number of options at any given decision point (see [1, 5, 3, 7–9] for examples on many different problems, including job shop scheduling, cutting/packing, and SAT). This is equivalent to an 'index policy' [10], because each potential option is given a score independently of other options, and the option with the largest score is selected.

In the example we present in this paper, the evolved mathematical expressions are used as policies for the online one-dimensional bin packing problem.

In this domain, the choices that are made using the expression can stay the same, even though the expression itself has changed, along with the values it produces for any given inputs. For example, an expression whose results are scaled by 2 gives the same relative scores to each option. In this situation, it is not enough to analyse the effects of genetic operators on the genotype, because this may not be the same as analysing the effects on the behaviour of the phenotype, which is what we are really interested in.

In this paper, we combine these issues with previous work in [14] and present a matrix analysis tool for understanding the effects of mutation on the phenotype. This can be used in any situation where the GP trees represent mathematical expressions with integer variables. The tool is based on the idea that a matrix can be generated from the expression by inputting all possible integer combinations, and storing the results. The resulting matrix will represent the expression exactly, as the only possible inputs are integers anyway. In this paper, we study integer variables, but we expect this to work for at least general discrete and finite cases.

A matrix can be generated for an expression before and after the mutation operator is applied, and the matrices can be compared to analyse what effect the mutation had. We show in this paper that many mutation calls return a different expression, but which corresponds to an equivalent matrix, and so its behaviour as a heuristic is the same. The proportion of such ineffective mutations varies during the run, but can be as high as 45%.

When an expression's matrix is different after a mutation, we measure how different, and find that the change is relatively large on average. A high proportion of the matrix is often modified by the mutation operator. With further analysis, this research could also provide insight into the code bloat phenomenon, as mutations become less effective as the generations increase. It also can be used as a tool for the analysis of GP runs, to check how the mutation is performing, and modifying its severity accordingly.

2 The Bin Packing Problem

The exact nature of the problem is a secondary concern in this paper, we are interested in analysing the mutation operator of the GP system. However, to do this, we need a problem domain for which to evolve heuristics, and the one dimensional bin packing problem is a highly appropriate domain to test on, given the volume of existing literature on evolving policies for this problem.

The one-dimensional bin-packing problem involves a set of integer-size pieces L, which must be packed into bins of a certain capacity C, using the minimum number of bins possible. In other words, the set of integers must be divided into the smallest number of subsets so that the sum of the sizes of the pieces in a subset does not exceed C [12]. We will assume that all of the bins have the same capacity, and that the pieces are drawn from a uniform distribution.

For this work we consider problem instances where 500 integer sized pieces are uniformly distributed in the range [5,10], and the bin capacity is 20. We use the following notation, UBP(20,5,10,500) to represent this domain. In this paper, the 'on-line' bin packing problem is studied. That is, we do not know in advance how many pieces there are or the size of those pieces. Our system must simply pack the pieces into the bins in the order they arrive, and the pieces cannot be moved once they have been placed in a bin.

3 Previous Work

GP was used to evolve heuristics for online one-dimensional bin packing in [4, 5]. In that work, the heuristics were expressions, which provided a score to each available bin. The GP system utilised the $+, -, *$, and % (protected divide, see [1]) operators, and the three terminals available to the GP were the piece size, the bin fullness, and the bin capacity. The current piece is put into the bin which received the highest score. No known heuristic has both a better worst case performance ratio and average uniform case performance ratio (with items drawn uniformly in the interval [0,1]) than the 'best-fit' heuristic [11], but heuristics were evolved which could beat best-fit on narrower distributions of pieces.

This work was later extended to reduce the number of inputs to two, as it was found that it was sufficient to use 'emptiness' (capacity - fullness) and piece size as GP terminals [6]. An extension has also been shown to utilise a 'memory' component to learn to use the distribution of piece sizes [2].

Parkes and Özcan noted that for a bin packing problem where the pieces and bins have integer size, the possible inputs to the expression are discrete. Therefore, the expression can be represented by a matrix. They showed that matrices themselves can be evolved with a genetic algorithm [14]. However, in this paper, we employ standard GP to evolve trees (mathematical expressions), and use their matrix representation to analyse the effects of the mutation operator. The matrix representation is explained in detail in section 4.

The key idea of this paper is that when evolving expressions with tree-based GP, a mutation could be made on a part of the tree which represents inactive code. While the tree would look different, the actual results returned by the tree would be the same for any given input values. Furthermore, the values returned by the tree are used to rank the bins by the scores that they receive, so the actual values do not matter. It is only the relative order of the scores that makes a difference. For this reason, a mutation could cause a change in the tree, which does cause a change in the results of the tree, but which *does not* cause a change in the behaviour of the tree when applied to the problem. A simple example of this is a mutation which adds 10 to the value returned by the tree. The tree would be modified, and the values returned would be modified, but the policy that the tree represents would *not* be modified. This is because the tree will always give the same bin the highest score (and the second highest, and so on).

4 The Matrix Representation

Figure 1 shows an example GP tree, with two inputs S and E. To choose a bin for a given piece, this tree is evaluated once for each available bin, and the bin with the highest score receives the piece. This system is presented in [4, 5], and is also employed in this paper.

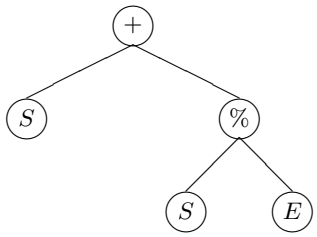

Fig. 1. An example heuristic function. S represents the piece size, E represents the emptiness of the bin (capacity-fullness)

Figure 2 shows the matrix which represents the expression in figure 1. The rows represent the remaining space left in a bin. The columns represent the size of the piece that we are currently choosing a bin for. The policy matrix contains heuristic values. The values are obtained by evaluating the expression (tree) with the remaining space and piece size as inputs. They show the heuristic score that is given to any bin by the tree, for any given piece size.

The dots in the matrix are positions which will never be used for an instance of UBP(20,5,10,500), as there are no piece sizes less than 5 and greater than 10. No bin can have a remaining space of between 16 and 19 (inclusive), because the minimum piece size is 5. These positions in the matrix are referred to as 'inactive' positions. We can calculate a value for them by evaluating the tree with the relevant inputs, but they will never be used. The other positions are referred to as 'active' positions.

If we add 10 to each of the values in figure 2, then it will make no difference to which bin receives the highest score. For a piece size of 5, a bin with 5 units of remaining space will always be chosen. For this reason, the exact values are not important, it is the relative order of the values that is important. Figure 3 shows a normalised version of the matrix in figure 2, where the bins are still ranked in the same order for a given piece size. In this paper, we are interested in whether the mutation operator makes changes in the normalised matrix, as this determines the behaviour of the heuristic.

To further clarify, the matrixes are used in the following manner. For any given piece size, there will be a column of scores, which correspond to the preference of which bin to put the piece into. For example, in figure 2, if we must pack a piece of size 6 next, then the piece will be put into a bin with an emptiness of 6, as this emptiness has the highest score (7.00) in that column of the matrix. However, if no available bin has an emptiness of 6, then the piece will

Fig. 2. Value matrix generated from the expression in figure 1

5	6.00	.	.	.	.	.
6	5.83	7.00	.	.	.	.
7	5.71	6.86	8.00	.	.	.
8	5.63	6.75	7.88	9.00	.	.
9	5.56	6.67	7.78	8.89	10.00	.
10	5.50	6.60	7.70	8.80	9.90	11.00
11	5.45	6.55	7.64	8.73	9.82	10.91
E 12	5.42	6.50	7.58	8.67	9.75	10.83
13	5.38	6.46	7.54	8.62	9.69	10.77
14	5.36	6.43	7.50	8.57	9.64	10.71
15	5.33	6.40	7.47	8.53	9.60	10.67
16	.	.	.	.	.	.
17	.	.	.	.	.	.
18	.	.	.	.	.	.
19	.	.	.	.	.	.
20	5.25	6.30	7.35	8.40	9.45	10.50
	5	6	7 S	8	9	10

Fig. 3. Normalised matrix generated from the matrix in figure 2

5	12	.	.	.	.	.
6	11	11	.	.	.	.
7	10	10	10	.	.	.
8	9	9	9	9	.	.
9	8	8	8	8	8	.
10	7	7	7	7	7	7
11	6	6	6	6	6	6
E 12	5	5	5	5	5	5
13	4	4	4	4	4	4
14	3	3	3	3	3	3
15	2	2	2	2	2	2
16	.	.	.	.	.	.
17	.	.	.	.	.	.
18	.	.	.	.	.	.
19	.	.	.	.	.	.
20	1	1	1	1	1	1
	5	6	7 S	8	9	10

be put into the bin with emptiness 7, as this has the second highest score and is therefore the second preference. If no available bin has a remaining space of 7, then the third preference is selected for the piece, and so on. An empty bin is always available, so the piece will always be able to be put there, even if none of the other preferences are available. The normalised matrix (figure 3) shows this rank ordering of bin emptiness values.

5 Genetic Programming Parameters

In this paper, we analyse 50 runs of a GP system with the parameters shown in table 1. We use the ECJ (Evolutionary Computation in Java) system, which is a mature and well known toolkit for genetic algorithms. We do not make any claims about the quality of these parameters, we chose them because we wish to analyse the mutation operator in the standard ECJ distribution.

For the reader interested in technical implementation details, the mutation analyser was implemented as an extension of the 'MutationPipeline' class, overriding the 'produce' method. The custom produce method converts the old individual and the new individual into their matrix form, and then compares them using the distance metrics described in section 6.

The population size is set to 10000, to ensure that there are a significant number of mutations in each generation. The results do not include those where the mutation fails and the parent is just copied. There must be an actual mutation for the change to be logged. The mutation can fail for example if the newly generated subtree cannot fit into the designated mutation point because

Table 1. The GP parameters

Population Size	10000
Generations	100
Crossover Probability	0.85
Reproduction Probability	0.05
Mutation Probability	0.1
Selection Method	Tournament Size 70
Initialisation Method	Ramped half-and-half
Initial Min and Max Tree Depth	2, 6
Max Tree Depth After Mutation	17

it violates the depth limit of 17. This is a standard default parameter in the ECJ system, which prevents very deep trees from forming. The standard tournament size is set to 7, for the standard ECJ population size of 1024. As we are using a larger population size of 10000, we increase the tournament size to 70.

The mutation operator is subtree mutation. The selection of a node is done probabilistically, with a 0.1 probability of selecting a terminal, and 0.9 probability of selecting a non-terminal node. The subtree is replaced by the 'Grow' method [15].

6 Distance Metrics

To analyse the effects of mutation on the policies, we must define a distance metric to measure the distance between two matrices. In this paper we employ three distance metrics, all of which operate on the normalised versions of the matrices. Only the active parts (see Section 4) of the matrices are included in the distance calculations. To illustrate the three metrics, we use the simple example of UBP(11, 4, 5). With bins of 11 capacity, and pieces between 4 and 5 inclusive (For the results section, our experiments are performed on instances of UBP(20, 5, 10)). Two policy matrices for this problem are shown in figures 4 and 5. For each column, each heuristic value (one per emptiness value) represents a preference of where to put the piece. The highest value represents the first preference, the second highest represents the second preference, and so on.

6.1 Metric 1

This metric simply counts the number of preferences that are different in the normalised matrices. In the example of figures 4 and 5, this metric would give the two matrices a difference value of 6. For the problem instances we address in this paper, the matrices are larger, and so a score of 57 means the two matrices are completely different. A score of 0 means that the two matrices are identical.

Fig. 4. Example Matrix A

	4	2	.
	5	4	4
	6	1	3
E	7	5	2
	8	.	.
	9	.	.
	10	.	.
	11	3	1
		4	5
		S	

Fig. 5. Example Matrix B

	4	3	.
	5	4	1
	6	1	2
E	7	5	3
	8	.	.
	9	.	.
	10	.	.
	11	2	4
		4	5
		S	

6.2 Metric 2

Recall that for any given piece size, there will be a column of heuristic values, which correspond to the preference of which available residual space to put the piece into. For example, consider the first columns from both matrices in figures 4-5. These columns represent the preference order for the piece size 4. We show the columns here as rows (excluding the inactive positions):

Column 1 of matrix 1: 2, 4, 1, 5, 3
Column 1 of matrix 2: 3, 4, 1, 5, 2

We see that for both columns, the preference is to put the piece into a bin with a residual capacity of 7, as this has the highest value (5). For both columns, if a bin does not exist with 7 units of space left, the second choice of bin would have a residual capacity of 5. We can write the preference order of residual capacities like this:

Residual capacity preference order 1: 7, 5, 11, 4, 6
Residual capacity preference order 2: 7, 5, 4, 11, 6

To calculate metric 2 we iterate through each preference order list. We add one point of similarity if both matrices have the same first preference. Then we move to the second preference and add one point if that is the same, and so on. For each column, we stop when the current preference is different, and move to the next column. In the example, 7 and 5 are ranked in the same order in both lists, and the next entries are different, so these columns have a similarity of 2. The columns for the piece size 5 have no similarity, as the first preferences are different (residual capacity 5 in the first matrix, and 11 in the second matrix). Therefore, according to metric 2, these matrices have a similarity of 2.

In the problem instances we use in this paper, the maximum score is 57, as there are 57 active positions to compare. We subtract the result from 57 to put the metric on the same scale as metric 1. Therefore, a score of 57 means that the two matrices have a different first choice for every piece. A score of 0 means that the two matrices are identical.

6.3 Metric 3

This is an ordering based metric, which involves comparing the columns of the matrices in a similar way to metric 2. While metric 2 asks how many elements of the preference order are the same (until the first difference), metric 3 asks how many of the elements that follow each preference are the same, regardless of their order. We calculate a preference order for each column, the same as we calculated for metric 2. This is shown again here for the first column, for ease of reference.

Residual capacity preference order 1: 7, 5, 11, 4, 6
Residual capacity preference order 2: 7, 5, 4, 11, 6

We iterate through each preference from order 1, and find the identical preference in order 2. For every value which follows that preference in both orders, we add one point of similarity. In our example:

7 precedes 5, 11, 4 and 6 in both lists (similarity of 4).
5 precedes 11, 4, and 6 in both lists (similarity of 3).
11 precedes 6 in both lists (similarity of 1).
4 precedes 6 in both lists (similarity of 1).

The total similarity for the first column is therefore 9 (4+3+1+1). The similarity of a column is subtracted from n(n-1)/2, as this is the maximum similarity score, where n is the length of a column. We perform the same calculation for the other columns. In the instances used in this paper, Metric 3 has a minimum value of 0, representing identical matrices, and a maximum value of 251.

This metric considers each preference order as a permutation, and tells us how many swaps would be needed between any two adjacent preferences, to get from one permutation to the other.

7 Results

This section presents our results, showing the effect of the mutation operator on the policy matrices, over the 100 generations. The calculations are based on 50 runs of the GP algorithm. For example, the results for generation 6 are calculated from all 50 sets of recorded values at generation 6.

In figure 6, first consider the dotted line, which represents the 'metric 1' difference between the matrices after mutation. In each generation, all of the mutations are measured with metric 1, and the average difference is calculated. A higher value means that the mutation operator has a larger effect on the policies. The plot shows that in the first 3 generations, the effect of mutation increases. After generation 3, the effect of mutation decreases gradually.

The second line in figure 6 shows the same calculation, but excluding the mutations that cause absolutely no change to the normalised matrix. One can see that when the mutation *does* make a change to the policy, that change

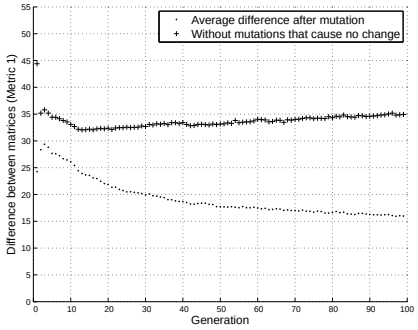

Fig. 6. Effect of mutation as measured by metric 1. The two plots show the average effects, and those not including the mutations that cause no change.

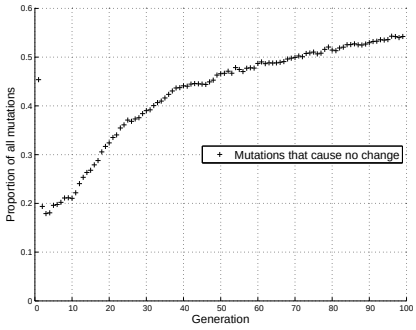

Fig. 7. Proportion of mutations per generation that cause no change in the normalised matrix, and therefore cause no change in the packing policy

is generally greater in the first few generations. That change decreases until generation 13, after which the change caused by the mutation operator gradually increases.

Figure 7 shows the proportion of mutations which do not change the normalised matrix at all, and therefore do not change the behaviour of the heuristic. This plot shows that in the first generation, around 45% of mutations have no effect on the individuals. This drops by a large amount in generation 2, to around 19%. This shows that the effect of mutation changes dramatically in the first 2 generations, as the population changes from a randomly generated one, to one made from the parents of the first generation.

Code bloat could be the cause of the downward trend in the effect of mutation. As code bloat increases the proportion of the tree which has no effect (See 'removal bias' theory [16] and 'replication accuracy' theory [13]), so the mutation operator is more likely to mutate a subtree which has no effect anyway. Of the mutations that do have an effect, the downward trend in the first 13 generations, followed by the gradual increase, is an effect which requires further research. We suspect that it involves the convergence of the population. For the one-dimensional bin packing problem, it could be the case that the population has generally converged to a good solution by around generation 13. If this is the case, then the results show that the mutation operator makes smaller changes as the population is improving. We would argue that this is a desirable quality, as it will make incremental changes to the better policies in later generations, while not changing the core functionality of the policy. Once the GP algorithm is generally improving the best individual less frequently, and the population is more stagnant, code bloat becomes a larger factor.

Figure 8 shows the results calculated with metric 2, which follow the same pattern, but the difference is measured as larger than metric 1. This could suggest

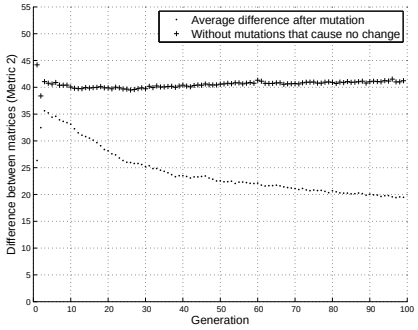

Fig. 8. Effect of mutation as measured by metric 2. The two plots show the average effects, and those not including the mutations that cause no change.

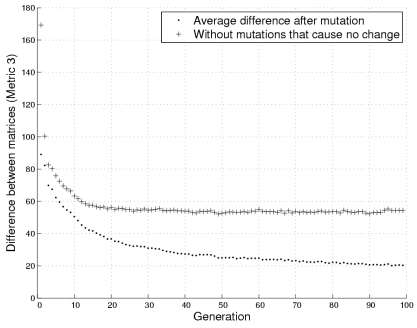

Fig. 9. Effect of mutation as measured by metric 3. The two plots show the average effects, and those not including the mutations that cause no change.

that the mutation operator often modifies the first few choices for each piece size, but not the lower choices (for example the 10th and 11th choice of bin).

Figure 9 shows the results calculated by metric 3. Recall that this metric considers how many preferences that follow each preference are the same, regardless of the order of the preferences. It tells us how many swaps would be needed to get from one preference permutation to the other. This metric suggests that, in later generations, the mutation operator changes less the relative preference order. In the early generations, the mutation operators make very large changes in the relative ordering of the bin preferences. The effect measured by metric 3 consistently decreases throughout the run, unlike the other two metrics.

When a mutation swaps two preferences that are far apart in the preference order, metric 3 measures a larger change. For example, if the 1st and 6th preferences swap, then metric 3 measures a larger change than if the 1st and 2nd preferences swap. In contrast, metric 1 measures the same change for both of those examples, as only two preferences are different. From the results of metric 3, we can infer that the changes later in the run are more localised, and that there is no significant further reduction in the severity of mutation after generation 15.

Figures 6-9 show the results per generation, but it is also interesting to consider the distribution of the mutation effects, rather than just the mean averages. Figures 10 and 11 show histograms of all the mutations from all generations in all 50 runs, not including the mutations which cause no change. Figure 10 shows the mutations measured with metric 2, and figure 11 shows the same mutations measured with metric 3. It is interesting to note that metric 2 measures the changes as mostly high, and metric 3 measures them as mostly low. From metric 2, we can say that the changes in the preference order mostly appear early in the preference order, which is the most influential part. By far, the most common difference value is 57, which represents a mutation which causes a change in the

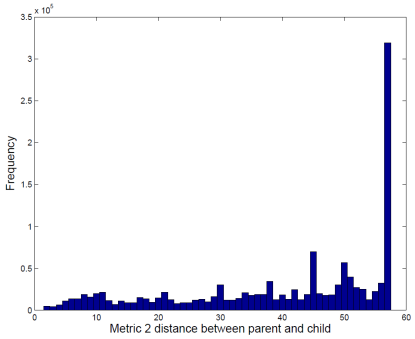

Fig. 10. Histogram of metric 2 effect of mutation.

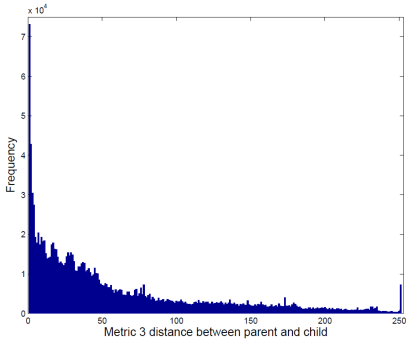

Fig. 11. Histogram of metric 3 effect of mutation.

policy where, for each piece, the first choice of bin will be modified. Even though the first choices often change, figure 11 shows that in general the mutations represent just a few swaps of adjacent preferences. The difference between figures 10 and 11 show that the magnitude of the mutation depends upon which metric is used to measure it.

8 Conclusions

We have presented a method for analysing the effects of the GP mutation operator in a normal GP run. It is applicable whenever the individual is a mathematical expression, with integer variables. Because of the integer input, the expression can be represented as a matrix of values, representing the value returned for each possible combination of inputs. In this study, the individuals were tree-based expressions which acted as index policies for the online bin packing problem. They gave a heuristic score to each bin, and the highest scoring bin received the next piece. The individuals therefore acted as policies for the handling of pieces when they arrive, depending on the bins available at the time.

The results show that the mutation operator causes large changes in the behaviour of the policies. It is generally not an operator that makes small changes to improve a policy incrementally. Of course, some of the changes were small, but the majority of mutations have a larger effect.

Over the course of the run, the effect of the mutation operator changes. Its effect was larger at the beginning of the run, and reduces as the run progresses. Our results indicate that more research is necessary into the the role of code bloat in reducing the effect of mutation. The 'replication accuracy' theory of code bloat [13] states that fit individuals which are large enough to not be affected by mutation are more likely to survive in the population, as their behaviour is unchanged from one generation to the next. The matrix analysis methodology could be employed to test the effect of mutation on the phenotype of large and

small individuals. In the future, we also plan to use the matrix analysis tool to analyse the effects of the crossover operator on the individuals.

There are many areas where this type of analysis tool could be used. For example, it could be used during a GP run to vary the severity of mutation depending on the effect it is having on the population. It could be used to assess the effectiveness of bloat control methods, which usually operate simply on the size of the trees. The analysis method presented here offers the chance to measure bloat by the effects of the mutation operator. Once developed further, this technique could be a valuable tool for effective parameter setting of the genetic operators in GP.

References

1. Allen, S., Burke, E.K., Hyde, M.R., Kendall, G.: Evolving reusable 3D packing heuristics with genetic programming. In: Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO '09). pp. 931–938. Montreal, Canada (July 2009), http://www.cs.nott.ac.uk/ mvh/papers/mvhgecco2009.pdf
2. Burke, E.K., Hyde, M., Kendall, G.: Providing a memory mechanism to enhance the evolutionary design of heuristics. In: Proceedings of the IEEE World Congress on Computational Intelligence (WCCI'10). pp. 3883–3890. Barcelona, Spain (July 2010), http://www.cs.nott.ac.uk/ mvh/papers/mvhcec2010.pdf
3. Burke, E.K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., Woodward, J.: Exploring hyper-heuristic methodologies with genetic programming. In: Mumford, C., Jain, L. (eds.) Computational Intelligence: Collaboration, Fusion and Emergence, pp. 177–201. Springer (2009), http://www.cs.nott.ac.uk/ mvh/papers/mvhGPasHH.pdf
4. Burke, E.K., Hyde, M.R., Kendall, G.: Evolving bin packing heuristics with genetic programming. In: Runarsson, T., Beyer, H.G., Burke, E., J.Merelo-Guervos, J., Whitley, D., Yao, X. (eds.) LNCS 4193, Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN'06). pp. 860–869. Reykjavik, Iceland (September 2006), http://www.cs.nott.ac.uk/ mvh/papers/mvhppsn2006.pdf
5. Burke, E.K., Hyde, M.R., Kendall, G., Woodward, J.: Automatic heuristic generation with genetic programming: Evolving a jack-of-all-trades or a master of one. In: Proceedings of the 9th ACM Genetic and Evolutionary Computation Conference (GECCO'07). pp. 1559–1565. London, UK. (July 2007), http://www.cs.nott.ac.uk/ mvh/papers/mvhgecco2007.pdf
6. Burke, E.K., Hyde, M.R., Kendall, G., Woodward, J.: The scalability of evolved on line bin packing heuristics. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC'07). pp. 2530–2537. Singapore (September 2007), http://www.cs.nott.ac.uk/ mvh/papers/mvhcec2007.pdf
7. Burke, E.K., Hyde, M.R., Kendall, G., Woodward, J.: A genetic programming hyper-heuristic approach for evolving two dimensional strip packing heuristics. IEEE Transactions on Evolutionary Computation 14(6), 942–958 (2010), http://www.cs.nott.ac.uk/ mvh/papers/mvh-draft-ieeetec2010.pdf
8. Fukunaga, A.S.: Automated discovery of local search heuristics for satisfiability testing. Evolutionary Computation (MIT Press) 16(1), 31–61 (2008)

9. Geiger, C.D., Uzsoy, R., Aytug, H.: Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. Journal of Scheduling 9(1), 7–34 (2006)
10. Gittins, J.C.: Bandit processes and dynamic allocation indices. Journal of the Royal Statistical Society. Series B (Methodological) 41(2), 148–177 (1979)
11. Kenyon, C.: Best-fit bin-packing with random order. In: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 359–364 (1996)
12. Martello, S., Toth, P.: Lower bounds and reduction procedures for the bin packing problem. Discrete Applied Mathematics 28(1), 59–70 (1990)
13. McPhee, N.F., Miller, J.D.: Accurate replication in genetic programming. In: Eshelman, L. (ed.) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95). pp. 303–309. Morgan Kaufmann, Pittsburgh, PA, USA (15-19 1995), citeseer.ist.psu.edu/mcphee95accurate.html
14. Özcan, E., Parkes, A.J.: Policy matrix evolution for generation of heuristics. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation. pp. 2011–2018. GECCO '11, ACM, New York, NY, USA (2011)
15. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. lulu.com, freely available at http://www.gp-field-guide.org.uk (2008)
16. Soule, T., Foster, J.A.: Removal bias: a new cause of code growth in tree based evolutionary programming. In: 1998 IEEE International Conference on Evolutionary Computation. pp. 781–786. Anchorage, Alaska, USA (5-9 May 1998), http://citeseer.ist.psu.edu/313655.html