A Relaxed Approach to Simplification in Genetic Programming

Mark Johnston[1], Thomas Liddle[1], and Mengjie Zhang[2]

[1] School of Mathematics, Statistics and Operations Research
[2] School of Engineering and Computer Science
Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand
{mark.johnston,liddlethom}@msor.vuw.ac.nz, mengjie.zhang@ecs.vuw.ac.nz

Abstract. We propose a novel approach to program simplification in tree-based Genetic Programming, based upon numerical relaxations of algebraic rules. We also separate proposal of simplifications from an acceptance criterion that checks the effect of proposed simplifications on the evaluation of training examples, looking several levels up the tree. We test our simplification method on three classification datasets and conclude that the success of linear regression is dataset dependent, that looking further up the tree can catch ineffective simplifications, and that CPU time can be significantly reduced while maintaining classification accuracy on unseen examples.

1 Introduction

One problem that limits the effective application of Genetic Programming is *program bloat* [1][2][3][4][5][6][7][8], where program trees tend to grow in size over the generations, causing the GP process to be computationally expensive. Bloat may arise from "model overfitting" (formulating a model that is more complicated than necessary to fit a set of training examples) but equally may occur with no fitness benefit. In addition, program trees sometimes appear contrived to make the best use of the available constant values set in the initial population. Several methods have been proposed to combat bloat: setting a maximum depth or number of nodes of a GP tree [1][4][9]; modifying the fitness function to reward smaller programs (parsimony pressure) [10][11][12]; dynamically creating fitness holes [5]; and operator equalisation [3].

In tree-based GP, program trees in the population may exhibit some algebraic redundancy, i.e., the mathematical expressions that the trees represent can often be directly mathematically simplified during the evolutionary process. This was first proposed by Koza [1] with his editing operation. Two approaches to simplification of programs are the algebraic and numerical approaches. In the *algebraic* approach [13][14][15], the rules of algebra are used (in a bottom-up fashion) to directly simplify the mathematical expression that the tree represents. In the *numerical* approach [16][17], the evaluation of each of the set of training examples is examined to determine if particular subtrees can be approximated by a single constant, removed altogether, or replaced by a smaller subtree. This is similar to “lossy compression” of images and aims for a minimal effect upon the evaluation of training examples.

In this paper, we propose to split the process of simplification into two roles: *proposers* which propose a local change to the program tree; and an *acceptor* which evaluates the proposed local change and determines whether to accept or reject it. The novel aspects are that the proposers use numerical relaxations of algebraic simplification rules, including linear regression, and that the acceptor evaluates the effect of the proposed local change further up the tree. The overall research goal is to determine how simplification affects classification accuracy and computational effort for classification problems. In particular, we wish to balance the number and severity of simplifications proposed (reduction in tree size or wasted proposals that are not accepted) and the additional workload in evaluating them.

The remainder of this paper is structured as follows. Section 2 provides background on algebraic and numerical approaches to simplification in GP programs. Section 3 develops our new approach to simplification of GP programs based upon a relaxation of the algebraic rules and separating the roles of simplification proposer and simplification acceptor. Section 4 describes computational experiments on three datasets and Section 5 discusses the results. Finally, Section 6 draws some conclusions and makes recommendations for future research directions.

2 Algebraic and Numerical Approaches to Simplification

In this section we review some existing algebraic and numerical approaches to the simplification of a program in tree-based GP. We consider a simple GP system which includes the basic arithmetic operators ($+$, $-$, $\times$ and protected division %) together with an `ifpos` operator (which returns the middle child if the left child is positive, and otherwise returns the right child).

2.1 Algebraic Simplification

Algebraic simplification of a GP tree involves the *exact* application of the simple rules of algebra to nodes of the tree in order to produce a smaller tree representing an exactly equivalent mathematical expression. For example, for constants a and c and subtree B, we can replace the subtree $a \times (B \times c)$ with the subtree $b \times B$ where $b = a \times c$ is a new constant node. This can be implemented efficiently using hashing in the finite field $\mathbb{Z}_p$ for prime p [14,15]. The strength of this approach is that any proposed simplification has *no global* effect on the evaluation of any training example. The weakness is that the rules of algebra are applied exactly, i.e., there is no scope for approximate equivalence, nor equivalence across the domain of the training examples. There are also some algebraic simplifications that are difficult for a basic set of locally applied algebraic rules to recognise when applied in a bottom-up fashion.

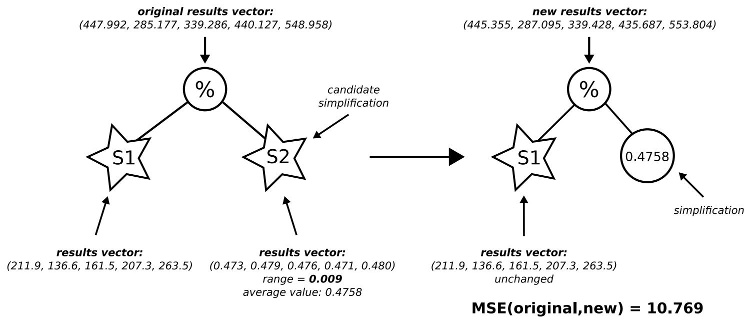


Fig. 1. An example where range simplification causes a (possibly) significant change to the tree one level up. The left subtree (`S1`) has relatively large values in its results vector (the evaluation of the subtree on the training examples), and is divided by the right subtree (`S2`) which has relatively small evaluation values. Even though the range of `S2` is only 0.009, the division means the simplification potentially magnifies the changes further up the tree.

2.2 Numerical Simplification

Numerical simplification of a GP tree involves the replacement of a subtree with a smaller (possibly *approximate*) substitute based upon the *local* effect on the evaluation of the training examples. Two simple methods recently investigated are:

1. *Range simplification* [16]. In evaluating the training examples, if the range of values a node takes is sufficiently small (less than a *range threshold*), then the node is replaced by a single constant-node (the average value). The strengths of range simplification are that equivalence is based only upon the observed range of the training examples; it also deals with nodes that are calculated from constant values; it allows for features or subtrees with a very small range of values to be simplified; and it is computationally inexpensive. However, the weakness is that local simplifications can have an adverse effect further up the tree in some cases. Figure 1 gives an indication of the potential effect of a local range simplification further up the tree. These changes may have a large effect on the outcome, but could otherwise be swamped by other sources of noise or uncertainty.
2. *Removing redundant children* [17]. In evaluating the training examples, if the difference between the values at a parent node and its child are sufficiently small (less than a *redundancy threshold* in this paper) then the parent can be replaced by the child. Song et al [17] use the criterion that the sum of absolute deviations (SAD) be zero over all training examples, i.e., $\sum_i |p_i - c_i| = 0$ where p_i and c_i are the evaluation of the ith training example at the parent and child respectively. This is a slight relaxation of algebraic simplification to the actual range of values taken by the training examples.

3 New Relaxed Approach to Simplification

We propose a new relaxed approach to simplification. Firstly, we use numerical evaluation of the training examples to determine if the algebraic rules are approximately satisfied. Secondly, we evaluate the numerical effect of any proposed local simplifications further up the tree before accepting them. Hence, we clearly separate the proposal of a local simplification from the acceptance or rejection of the proposal based upon its effect on the numerical evaluation of the training examples. This addresses the weakness of exact algebraic simplification by covering simple algebraic rules and allows for approximate satisfaction of these rules. It also addresses the weakness of local numerical simplification by looking at the effect further up the tree before accepting a proposed simplification.

3.1 Proposers

In this paper we use three numerical simplification operators — range simplification and removal of redundant children (as in Section 2.2), and linear regression (described further below) — to numerically evaluate possible algebraic simplifications, relaxing each equality slightly. Between the three operators, we cover most simple algebraic rules. We make a small modification to each of the first two operators presented before: for simplicity, we use a constant range threshold for range simplification; and we use mean square error (MSE) for redundancy checking (rather than SAD).

Linear regression. Consider the nodes Y and S in a GP tree, where S is a child or grandchild subtree of Y. If we can approximate Y by

$$Y = b \times S + a \tag{1}$$

or

$$Y = b \ \% \ S + a \tag{2}$$

sufficiently closely for some constants a and b, then we may be able to significantly reduce the size of the tree. This is an extension of simple algebraic rules and allows for *approximate linearity* of node Y against subtree S (or $\frac{1}{S}$). Figure 2 gives two examples in which linear regression will reduce a tree where other simplification methods do not. A candidate simplification's tree size using this method will be a maximum of $4 + |S|$ nodes, with a possible simplification to $2 + |S|$ under certain conditions on a and b, where $|S|$ is the number of nodes in subtree S. To evaluate linearity, we use Pearson's correlation coefficient. We consider all children and grandchildren of Y as S for simplification and choose the one with the highest value of Pearson's r^2 greater than a *regression threshold.* The proposal is to replace node Y by the simplest version of equation (1) or (2) as appropriate.

3.2 Acceptor

In order to check that a proposed simplification won't cause a significant change further up the tree, we compare the results vectors (the evaluation of the subtree

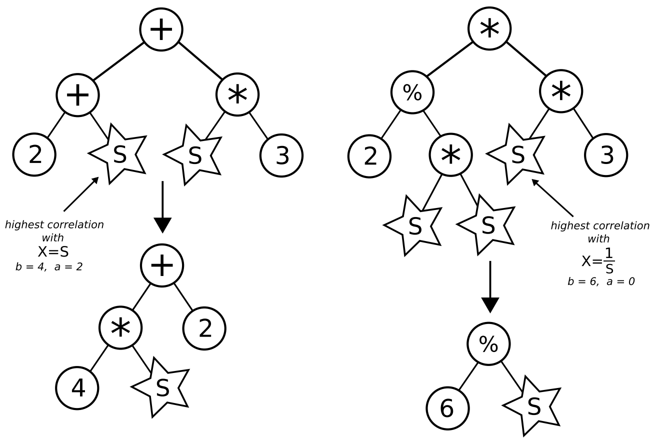


Fig. 2. Simplification examples that are not covered by simple (local) algebraic rules, but are covered by linear regression. Here S represents a particular repeated subtree in each example and Y corresponds to the entire subtree.

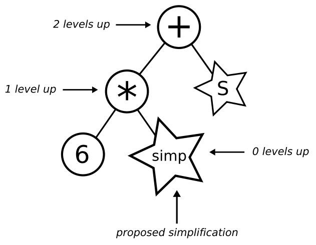


Fig. 3. The acceptor evaluates the effect of a proposed simplification n levels up the tree. Here, arrows point to the node that the MSE calculation applies.

on all training examples) of the old and new (simplified) tree. Figure 3 illustrates which nodes are checked against for different values of n. We go to the ancestor node n levels up and calculate the mean square error (MSE) at that node, i.e., $\sum_i (new_i - old_i)^2$, where old_i and new_i are the original and newly simplified evaluations of the ith training example respectively. If the MSE is less than an *acceptance threshold*, then we accept the simplification and make the change to the tree; if it is not, then we reject the simplification and keep the old tree. In this way we aim to change the tree's fitness as little as possible.

4 Experimental Design

Datasets. To test our simplification system we ran experiments on three different classification datasets: Coins (14 features, 3 classes, [14][16]), Wine (13 features, 3 classes, [18]) and Breast-Cancer Wisconsin (9 features, 2 classes, [19]). Coins consists of 600 images (each 64×64 pixels) of five cent pieces against a random noisy background. Wine gives the result of a chemical analysis of Italian wines from three cultivars (the classes). Each instance of the Breast-Cancer Wisconsin dataset corresponds to a benign or malignant diagnosis.

GP system setup. All experiments were run with the following setup: population size 100, number of generations 100, maximum depth of tree 40, mutation rate 28%, crossover rate 70%, elitism rate 2%. The terminal set consists of the features and random float numbers in the range $[-10, 10]$. We used *static range selection* [20] to choose the class from the tree output and ten-fold cross validation to evaluate each tree in the population.

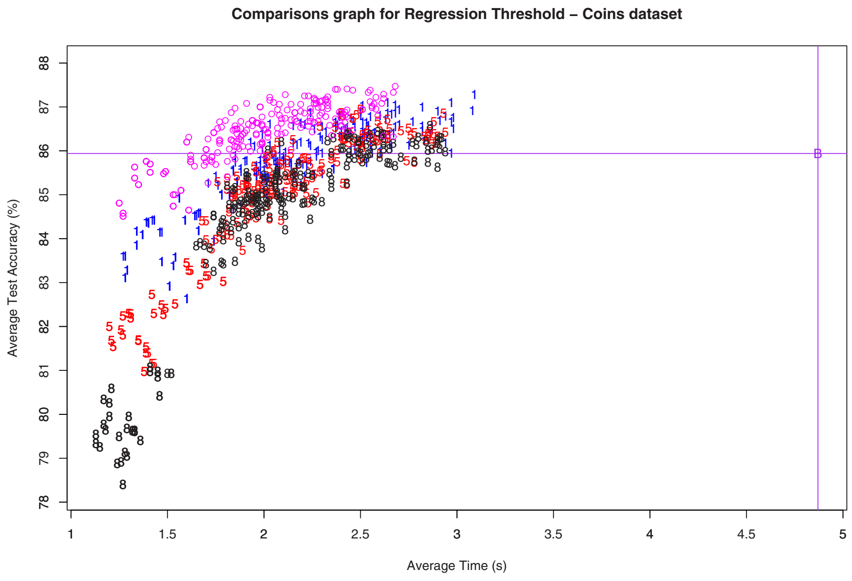
Simplification frequency. We perform simplification checks on the whole population every k generations, simplifying the population before the selection process occurs for the next generation. We do not simplify the initial population as this may remove too many of the useful "building blocks" present.

Choice of threshold values. For the operators we have implemented there are six different thresholds that we need to test in our experiments: the *proposal thresholds* (range width, redundant MSE and regression r^2); the *acceptance thresholds* (acceptance MSE and the number of levels to look up n); and simplifying the population every k generations. Preliminary experiments suggested a reasonable range of values of each threshold. The set of values for each threshold used in our more extensive experiments can be seen in Table 1, so considering all combinations we have $3^5 \times 4 = 972$ configurations in total, and we ran each configuration on the same set of 100 random seeds.

5 Results and Discussion

Classification accuracy vs computational effort. Table 1 summarises the results for each dataset. The base result is a standard GP with no simplification (and recall that the maximum tree depth is 40), for comparison with all other results. All datasets performed differently in our tests. Regarding average test accuracy, the Coins dataset fluctuated greatly over all configurations, some performing much worse than the base system, but some also a lot better (see Figure 4). On the other hand, the Wisconsin dataset's average test accuracy is virtually unchanged in the range $[95.22\%, 95.71\%]$, while the Wine dataset is at least 8–9% worse than the base system. When considering computational effort (CPU time), all datasets show significant savings. The biggest 'reasonable' time savings (meaning not too much degradation in test accuracy) for the Coins dataset is approximately 75% savings, Wisconsin 60%, and Wine 35%. The Wine dataset runs so quickly, however, that changes in CPU time are difficult to measure accurately, and the time taken across all configurations varies within approximately 0.1 of a second.

Proposal and acceptance thresholds. In general as we increase the value of each of the range width, redundant MSE and acceptance MSE thresholds, CPU time



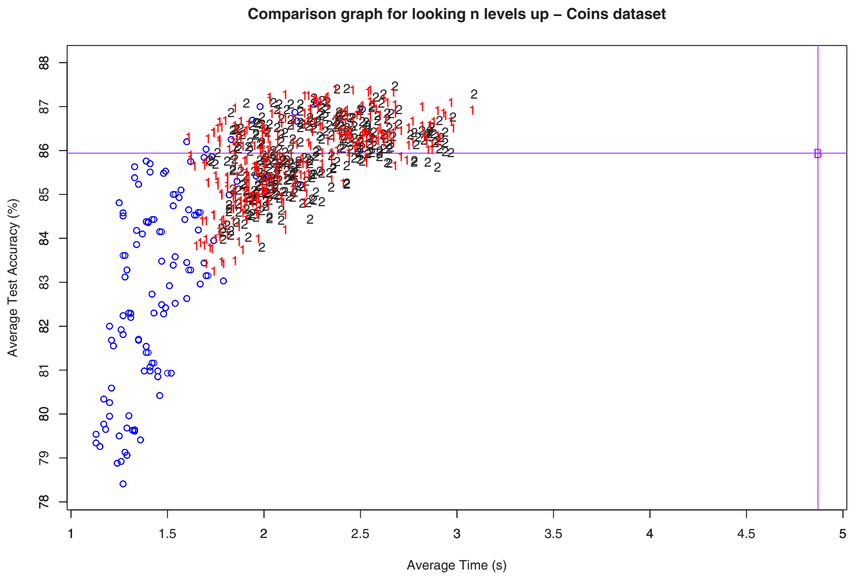


Fig. 4. Two scatter plots showing the average test accuracy vs average CPU time for the Coins dataset. Each point is one of the 972 configurations. The top graph highlights the different values for the regression threshold ('○' = no regression, '1' = 0.99, '5' = 0.95, and '8' = 0.80), and the bottom graph highlights looking n levels up. The lines represent the performance of the base system for comparison.

Table 1. Average CPU time taken (in seconds) and test classification accuracy (as a proportion) grouped by different thresholds for each dataset. Results for each of the three levels of the range threshold are collected over $3^4 \times 4 = 324$ combinations of the other five thresholds, etc.

	Coins				*Wine*				*Wisconsin*			
	Time	(s.d.)	T.Acc	(s.d.)	Time	(s.d.)	T.Acc	(s.d.)	Time	(s.d.)	T.Acc	(s.d.)
Base	4.87	1.80	0.8594	0.0314	1.09	0.40	0.7346	0.0379	6.66	2.27	0.9532	0.0063
Range Threshold												
0.1	2.07	0.25	0.8490	0.0205	0.70	0.02	0.6567	0.0305	3.93	0.55	0.9546	0.0020
0.5	1.96	0.22	0.8498	0.0202	0.70	0.02	0.6503	0.0295	3.89	0.53	0.9546	0.0021
1.0	1.89	0.20	0.8489	0.0192	0.70	0.02	0.6466	0.0287	3.84	0.52	0.9545	0.0021
Redundancy Threshold												
0.01	2.11	0.26	0.8540	0.0188	0.70	0.02	0.6531	0.0295	3.92	0.54	0.9546	0.0020
0.05	1.94	0.21	0.8491	0.0203	0.70	0.02	0.6507	0.0294	3.88	0.53	0.9546	0.0021
0.10	1.88	0.20	0.8445	0.0205	0.70	0.02	0.6498	0.0296	3.86	0.53	0.9546	0.0021
Regression Threshold												
none	1.96	0.26	0.8632	0.0210	0.70	0.02	0.6399	0.0295	3.42	0.57	0.9541	0.0024
0.99	2.07	0.25	0.8547	0.0220	0.71	0.02	0.6582	0.0314	4.22	0.61	0.9543	0.0022
0.95	1.96	0.21	0.8446	0.0201	0.70	0.02	0.6548	0.0300	4.04	0.53	0.9549	0.0021
0.80	1.90	0.19	0.8343	0.0175	0.70	0.02	0.6520	0.0280	3.86	0.47	0.9549	0.0018
Levels Up												
0	1.49	0.11	0.8301	0.0192	0.68	0.02	0.6241	0.0215	3.15	0.31	0.9550	0.0019
1	2.18	0.28	0.8585	0.0210	0.71	0.03	0.6630	0.0349	4.35	0.70	0.9543	0.0022
2	2.25	0.29	0.8590	0.0208	0.71	0.03	0.6665	0.0343	4.15	0.61	0.9543	0.0023
Acceptance Threshold												
0.01	2.21	0.29	0.8534	0.0193	0.70	0.02	0.6563	0.0306	3.98	0.56	0.9545	0.0021
0.05	1.90	0.20	0.8485	0.0199	0.70	0.02	0.6499	0.0296	3.87	0.53	0.9546	0.0020
0.10	1.80	0.18	0.8457	0.0201	0.70	0.02	0.6474	0.0285	3.81	0.51	0.9546	0.0020
Simply Every k Generations												
3	2.08	0.24	0.8487	0.0200	0.71	0.02	0.6585	0.0309	4.29	0.60	0.9546	0.0021
4	1.94	0.22	0.8490	0.0194	0.70	0.02	0.6488	0.0290	3.81	0.53	0.9546	0.0022
5	1.89	0.22	0.8499	0.0201	0.70	0.02	0.6463	0.0297	3.55	0.48	0.9545	0.0022

goes down (Wine stays constant however), but so does average test accuracy (except for Coins when the range threshold is 0.5 and Wisconsin which stays fairly constant). It appears that linear regression is causing more computational overhead than it is worth. The Coins dataset shows this most clearly (see the top graph in Figure 4): the time taken with no regression is similar to that with 0.95 and 0.80 values, but the test accuracy stays higher, i.e., additional computational overhead is not offset by the simplifications made. We see similar CPU time savings without regression in the Wisconsin dataset, but test accuracy remains fairly constant. On the Wine dataset, however, using linear regression has higher test accuracy than not using it, but the test accuracy is still significantly less than that of the base system.

How far up the tree to evaluate. In general it seems that as we increase the number of levels we look up before accepting a simplification, the overall average CPU time increases (with the exception of Wisconsin with 2 levels), but so does the test accuracy (Wisconsin's test accuracy remains relatively constant however). This is best displayed in the Coins dataset where both CPU time and test accuracy change significantly (see the bottom graph in Figure 4). In general, looking 0 levels up amounts to a significant time reduction but also a significant reduction in test accuracy, while looking 1 or 2 levels up only is slightly more computationally expensive but maintains a lot higher test accuracy.

How often to simplify. Overall, there doesn't seem to be much change in test accuracy among the different values for k. As we simplify less often, the CPU time reduces significantly on the Coins and Wisconsin datasets, while the time remains unchanged on Wine. This indicates that it might be useful to investigate simplifying even less often.

Comparing number of proposals vs number of acceptances. A central research question is how much computational overhead arises from generating proposals and testing for acceptance. We expect that relaxing the proposal thresholds generates more proposals, each of which must be tested for acceptance. Table 2 compares the number of simplifications proposed and accepted, and percentage accepted, for each proposal operator. It shows the effect of increasing the acceptance threshold within each of these for the Coins dataset (however, the following general observations apply across all datasets) as follows. Unexpectedly, we see *fewer* proposed simplifications (it is apparent that there may be some "repeat proposing" of simplifications, i.e., a candidate gets rejected but is proposed again later on since it is still a good candidate at the local level—this also explains the higher CPU time for more stringent acceptance threshold values). As expected, the number of accepted proposals increases (except for Coins when the regression threshold is in $\{0.99, 0.95\}$, where the number accepted is relatively similar for acceptance threshold in $\{0.05, 0.10\}$). The average percentage of proposals accepted also increases, although at different rates for each dataset and proposal operator (the best acceptance percentage was just over 50%). The CPU time decreases due to a combination of fewer proposals (lower calculation overhead) and higher number of proposals accepted (overhead incurred in our implementation if a simplification proposal is rejected). The Coins dataset shows the largest reduction in CPU time, while none is observed on the Wine dataset. As expected, the average test accuracy decreases—as we accept less accurate approximations of portions of the tree, this causes the tree itself to have poorer accuracy in general. Again, Wisconsin is an exception, showing little change in test accuracy. Coins shows the highest reduction in test accuracy as well as CPU time seen above, so there seems to be a tradeoff. It is interesting to note, however, that some individual combinations of the simplification operators actually increase the average test accuracy compared to the base system (see the top graph in Figure 4). This could mean that simplifications are taking place in an early generation, allowing more of the search space to be covered in less time, but further research would be required to establish this.

How the proposal thresholds affect the number of proposals and acceptances. Across all datasets, for the linear regression operator, decreasing the value of

Table 2. Comparison of number of simplifications proposed vs number accepted for the Coins dataset. Results for each of the three levels of the range threshold by three levels of the acceptance threshold are collected over $3^3 \times 4 = 108$ combinations of the other four thresholds, etc.

	Proposal Thresh.	Accept Thresh.	#Prop	#Acpt	%Acpt	Time (s)	(sd)	T.Acc	(sd)
	Base Sys.	-	-	-	-	4.87	1.8	0.8594	0.0314
Range Thresh.	0.1	0.01	468.37	131.54	28.09	2.29	0.52	0.8531	0.0218
		0.05	340.77	142.53	41.82	2.00	0.35	0.8483	0.0199
		0.10	301.51	147.29	48.85	1.91	0.32	0.8455	0.0193
	0.5	0.01	428.95	111.66	26.03	2.18	0.49	0.8535	0.0208
		0.05	305.13	120.69	39.55	1.89	0.31	0.8492	0.0192
		0.10	270.29	125.77	46.53	1.81	0.28	0.8467	0.0188
	1.0	0.01	422.51	95.31	22.56	2.17	0.65	0.8535	0.0223
		0.05	284.44	103.66	36.44	1.83	0.41	0.8481	0.0197
		0.10	228.95	96.33	42.07	1.68	0.31	0.8450	0.0186
Redundant Thresh.	0.01	0.01	486.03	136.64	28.11	2.27	0.51	0.8566	0.0190
		0.05	370.32	159.29	43.01	2.07	0.38	0.8537	0.0180
		0.10	326.42	162.73	49.85	1.98	0.35	0.8517	0.0175
	0.05	0.01	436.42	109.46	25.08	2.20	0.57	0.8537	0.0215
		0.05	291.89	111.38	38.16	1.84	0.31	0.8477	0.0188
		0.10	255.86	114.56	44.78	1.77	0.27	0.8460	0.0183
	0.10	0.01	397.38	92.41	23.26	2.16	0.60	0.8498	0.0235
		0.05	268.14	96.21	35.88	1.80	0.34	0.8442	0.0207
		0.10	218.47	92.10	42.16	1.66	0.24	0.8394	0.0188
Regression Thresh.	none	0.01				2.13	0.42	0.8653	0.0076
		0.05	–	–	–	1.90	0.30	0.8629	0.0066
		0.10				1.83	0.29	0.8614	0.0066
	0.99	0.01	252.68	78.89	31.22	2.31	0.56	0.8580	0.0124
		0.05	181.62	81.38	44.81	2.00	0.37	0.8539	0.0100
		0.10	157.63	81.11	51.46	1.90	0.32	0.8520	0.0093
	0.95	0.01	517.82	142.15	27.45	2.22	0.59	0.8500	0.0201
		0.05	365.83	151.47	41.40	1.89	0.37	0.8439	0.0161
		0.10	313.79	151.09	48.15	1.78	0.30	0.8399	0.0138
	0.80	0.01	989.28	230.30	23.28	2.18	0.65	0.8401	0.0300
		0.05	693.01	256.32	36.99	1.81	0.40	0.8333	0.0254
		0.10	596.26	260.33	43.66	1.70	0.32	0.8296	0.0231

the threshold increases the number of simplification proposals. Proportionately, the number of proposals accepted increases but on average the percentage of proposals accepted decreases, indicating the acceptance operator is working. Surprisingly, for range simplification and redundancy, as we increase the value of the threshold, we actually see a *reduction* in proposals on average, and therefore a reduction in proposals accepted as well. A possible reason for this reduction could be the nature of the proposal operators: in both cases, once a simplification has occurred, those nodes can no longer be further simplified through these two methods. However, a linear regression simplification could in turn allow for another simplification the next level up the tree, a sort of cascading effect.

6 Conclusions

All configurations of the simplification operators significantly reduced the CPU time for the GP process to run. However, the tradeoff between CPU time and classification accuracy was different for different configurations and different datasets. Range simplification and removing redundant children appear to be useful simplification operators to use because they are simple and computationally efficient. However, the computational tests were inconclusive as to whether the linear regression operator we introduced is worth using (good for Coins, poor for Wine, no change for Wisconsin). Evaluating the effect of proposed simplifications further up the tree (rather than blind acceptance) appears to be very effective (Coins and Wine show that classification accuracy improves); looking one level up seems to be sufficient. As there is little reduction in test accuracy for any of the acceptance MSE threshold values tested in this paper, a more lenient MSE value may be desired for further CPU time reductions. Finally, when simplifying a population, it seems to be better to do so less often because of the high overhead incurred, so the less often you simplify, the faster the GP process runs (our best results were simplifying every five generations).

Avenues for future research include investigating the effect of simplification on tree size and tree depth across different generations, eliminating repeat proposal of the same simplification by the regression operator, applying the linear regression operator on more datasets to see if there is any consistency amongst different types of problems, and further investigating simplifying less often to find the optimal balance between size reduction and computational overhead.

Acknowledgment

This work was supported in part by the University Research Fund (URF09-2399/85608) at Victoria University of Wellington and by the Marsden Fund (08-VUW-014) administrated by the Royal Society of New Zealand.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
2. Blickle, T., Thiele, L.: Genetic programming and redundancy. In: Hopf, J. (ed.) Genetic Algorithms within the Framework of Evolutionary Computation, Max-Planck-Institut für Informatik (MPI-I-94-241), pp. 33–38 (1994)
3. Dignum, S., Poli, R.: Operator equalisation and bloat free GP. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 110–121. Springer, Heidelberg (2008)
4. Dignum, S., Poli, R.: Crossover, sampling, bloat and the harmful effects of size limits. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 158–169. Springer, Heidelberg (2008)

5. Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 204–217. Springer, Heidelberg (2003)
6. Soule, T., Foster, J.A., Dickinson, J.: Code growth in genetic programming. In: Koza, J.R., et al. (eds.) Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, pp. 215–223. MIT Press, Cambridge (1996)
7. Soule, T., Heckendorn, R.B.: An analysis of the causes of code growth in genetic programming. Genetic Programming and Evolvable Machines, 283–309 (2002)
8. Streeter, M.J.: The root causes of code growth in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 443–454. Springer, Heidelberg (2003)
9. Crane, E.F., McPhee, N.F.: The effects of size and depth limits on tree based genetic programming. In: Yu, T., et al. (eds.) Genetic Programming Theory and Practice III. Genetic Programming, vol. 9, pp. 223–240. Springer, Heidelberg (2005)
10. Nordin, P., Banzhaf, W.: Complexity compression and evolution. In: Eshelman, L. (ed.) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA 1995), Pittsburgh, PA, USA, pp. 310–317. Morgan Kaufmann, San Francisco (1995)
11. Zhang, B.T., Mühlenbein, H.: Balancing accuracy and parsimony in genetic programming. Evolutionary Computation 3(1), 17–38 (1995)
12. Luke, S., Panait, L.: Lexicographic parsimony pressure. In: Langdon, W.B., et al. (eds.) Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO 2002), New York, pp. 829–836. Morgan Kaufmann, San Francisco (2002)
13. Hooper, D., Flann, N.S.: Improving the accuracy and robustness of genetic programming through expression simplification. In: Koza, J.R., et al. (eds.) Genetic Programming 1996: Proceedings of the First Annual Conference, p. 428 (1996)
14. Wong, P., Zhang, M.: Algebraic simplification of GP programs during evolution. In: Keijzer, M., et al. (eds.) Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO 2006), Seattle, Washington, USA, July 8–12, vol. 1, pp. 927–934. ACM Press, New York (2006)
15. Zhang, M., Wong, P., Qian, D.: Online program simplification in genetic programming. In: Wang, T.-D., Li, X., Chen, S.-H., Wang, X., Abbass, H.A., Iba, H., Chen, G.-L., Yao, X. (eds.) SEAL 2006. LNCS, vol. 4247, pp. 592–600. Springer, Heidelberg (2006)
16. Kinzett, D., Zhang, M., Johnston, M.: Using numerical simplification to control bloat in genetic programming. In: Li, X., Kirley, M., Zhang, M., Green, D., Ciesielski, V., Abbass, H.A., Michalewicz, Z., Hendtlass, T., Deb, K., Tan, K.C., Branke, J., Shi, Y. (eds.) SEAL 2008. LNCS, vol. 5361, pp. 493–502. Springer, Heidelberg (2008)
17. Song, A., Chen, D., Zhang, M.: Bloat control in genetic programming by evaluating contribution of nodes. In: Raidl, G., et al. (eds.) Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO 2009), Montreal, pp. 1893–1894. ACM, New York (2009)
18. Forina, M., Leardi, R., Armanino, C., Lanteri, S.: PARVUS: An Extendable Package of Programs for Data Exploration, Classification and Correlation. Elsevier, Amsterdam (1988)
19. Asuncion, A., Newman, D.J.: UCI Machine Learning Repository (2007), `http://www.ics.uci.edu/~mlearn/MLRepository.html`
20. Zhang, M., Ciesielski, V.: Genetic programming for multiple class object detection. In: Foo, N.Y. (ed.) AI 1999. LNCS (LNAI), vol. 1747, pp. 180–192. Springer, Heidelberg (1999)