Noname manuscript No.
(will be inserted by the editor)

Numerical Simplification for Bloat Control and Analysis of Building Blocks in Genetic Programming

David Kinzett · **Mark Johnston** · **Mengjie Zhang**

the date of receipt and acceptance should be inserted later

Abstract In tree-based genetic programming, there is a tendency for the size of the programs to increase from generation to generation, a phenomenon known as bloat. It is standard practice to place some form of control on program size either by limiting the number of nodes or the depth of the program trees, or by adding a component to the fitness function that rewards smaller programs (parsimony pressure). Others have proposed directly simplifying individual programs using algebraic methods. In this paper, we add node-based numerical simplification as a tree pruning criterion to control program size. We investigate the effect of online program simplification, both algebraic and numerical, on program size and resource usage. We also investigate the distribution of building blocks within a genetic programming population and how this is changed by using simplification. We show that simplification results in reductions in expected program size, memory use and computation time. We also show that numerical simplification performs at least as well as algebraic simplification, and in some cases will outperform algebraic simplification. We further show that although the two online simplification methods destroy some existing building blocks, they effectively generate additional new and more diverse building blocks during evolution, which compensates for the negative effect of disruption of building blocks.

Keywords Genetic Programming, Program Simplification, Code Bloat, Building Blocks

David Kinzett
School of Engineering and Computer Science
Victoria University of Wellington, PO Box 600, Wellington, New Zealand
E-mail: kinzetalan@ecs.vuw.ac.nz

Mark Johnston
School of Mathematics, Statistics and Operations Research
Victoria University of Wellington, PO Box 600, Wellington, New Zealand
E-mail: mark.johnston@mso.vuw.ac.nz

Mengjie Zhang
School of Engineering and Computer Science
Victoria University of Wellington, PO Box 600, Wellington, New Zealand
E-mail: mengjie.zhang@ecs.vuw.ac.nz

1 Introduction

In tree based Genetic Programming (GP) there is a tendency for the size of programs to increase, a process known as *bloat* [6, 1–3]. This has a number of undesirable effects including increased memory usage and increased computation time. One of the main causes is the recombination via crossover. Standard practice is to give all nodes some chance of being the crossover point. Hence, as the programs become deeper (in levels), the average depth of the crossover point also becomes deeper, crossover is less likely to breed a child with a significantly different fitness, and the overall efficiency of the GP search is reduced.

A number of strategies have been proposed to combat bloat. Firstly, a limit can be placed on program size, limiting either the number of nodes or the depth of the tree [6,4,5]. This approach prevents bloat to some extent, but has some limitations as it is very difficult to set a good limit without prior knowledge. If the problem is not already well understood, then a series of trials will be required to establish what a good limit would be. This can be a significant extra computational cost in finding a solution. Also, when performing a crossover, trimming the subtrees being exchanged to fit the limit discards genetic material that may be important [4, 20].

Secondly, a component can be added to the fitness function, or the selection process, that rewards smaller programs, a practice known as *parsimony pressure* [6–11]. This approach can be very successful in many problems, as it goes some way to avoiding the loss of good programs that can occur with hard limits in that, if a program is good enough, then it will survive regardless of size. There are, however, situations when it can fail badly with all programs reducing to trivial sizes [13]. In fact, there are a number of reports on this approach that the effectiveness performance deteriorates [6, 7].

Thirdly, a program can be simplified algebraically during the run [20]. However evaluating whether two sub-trees are equivalent becomes computationally expensive as the size of the sub-trees increases. The algorithms necessary to address this are complex, and [20] uses efficient hashing techniques. This approach is however not easy to implement, and the hash algorithms are not infallible as like all hash techniques collisions are possible.

Our approach extends the program simplification approach by considering numerical simplification. In addition we examine what effect these two methods have on the distribution of building blocks within a population[1].

1.1 Research Goals

The goal of this research is to investigate a new program simplification approach to bloat control in tree based GP. We will consider two methods, one using simple algebraic simplification and the other with numerical simplification. We will compare the behaviour and performance of GP systems with both simplification methods and with no simplification. The specific questions we examine are:

1. Will the two simplification methods produce smaller programs than the canonical GP system with no simplification thereby reducing memory usage and shortening computation run times?
2. Will the simplification process affect the system effectiveness?

[1] When we use the term *building block* we mean a sub tree of the specified depth. It may occur at any point in the program of which it is a part.

3. How are the building blocks distributed, both within the search space, and as the evolution proceeds through the generations?
4. Do the two simplification methods change this distribution, and if so, do any changes depend on which of the two simplification methods is used?
5. Does the simplification process affect the overall diversity of building blocks within the population?

2 Simplification Methods

2.1 Algebraic Simplification

The aim of simplification is to remove "redundant" content from programs. The *algebraic* simplification approach [20] removes the redundancy by structurally changing the programs in such a way as to leave the programs functionally the same as before simplification. In other words, the new programs produced by the simplification process will produce exactly the same result as the original programs.

This method is motivated by the algebraic nature of programs using the canonical GP functions $(+, -, \times, \div)$ and uses a set of *algebraic simplification rules* to remove redundancies. These rules consist of two parts, a *precondition* which represents the state of the surrounding nodes that must be present for the rule to be applied, and a *postcondition* which represents the state that the surrounding nodes are in after additions and deletions are made. These rules make up the *rule-set* for the simplification method. Table 1 shows the particular rule-set used in this paper.

The method uses a hashing algorithm to simplify the check on sub-tree equality. Each operator has a hash value, and each feature has a hash value based on the feature number. Ephemeral constants have a hash value based on the value of the constant. These are combined in a "shift and xor" manner to generate the hash for their parent node. If two subtrees have the same hash value, they are considered equivalent. In our implimentation the hash calculations and rule evaluations are performed by the the *operator* objects. This means that the hash calculation can be aware of whether the operands need to be ordered or not, and that only those simplification rules that apply to that operator need to be checked.

To simplify a program, the rule-set is applied using a "greedy" engine. It recursively traverses the program tree in a postfix bottom-up fashion. For each node it processes in

Precondition		Result
$a+b$	$\rightarrow$	$c, c=a+b$
$a-b$	$\rightarrow$	$c, c=a-b$
$a\times b$	$\rightarrow$	$c, c=a\times b$
$a\div b$	$\rightarrow$	$c, c=a\div b$
$A\div 1$	$\rightarrow$	A
$A\div A$	$\rightarrow$	1
$0\div A$	$\rightarrow$	0
$0\times A=A\times 0$	$\rightarrow$	0
$A\times 1=1\times A$	$\rightarrow$	A
$A+0=0+A$	$\rightarrow$	A
$A-0$	$\rightarrow$	A
$A-A$	$\rightarrow$	0
`IfPos`(a, B, C) and a >0	$\rightarrow$	B
`IfPos`(a, B, C) and a ≤ 0	$\rightarrow$	C

Table 1 Simplification rules. Lower case letters represent numerical constants, while the upper case letters represent variable/feature terminal nodes or sub-trees.

this way, it checks the precondition of each rule in the rule-set. If any rule matches, then it is applied to that portion of the tree. The algorithm continues to check preconditions until none of the rules in the rule-set can be applied, in which case it moves on to the next node.

An example of the method is given in Figure 1. In this example, `F1` has been randomly assigned the hash value 6, while `F2` has been assigned the hash value 7. For the '+' nodes, the operator is aware that order is not important therefore the hash values calculated on the two '+' nodes will be the same. For the root node $\div$, the rule $A \div A \rightarrow 1$ is found to apply, since both left and right child nodes have the same hash value. The rule is applied and the result is a single numerical-node, with a value of `1`.

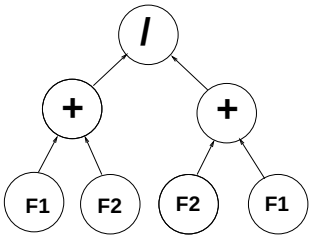


Fig. 1 An example of the algebraic simplification using the rule: $A \div A \rightarrow 1$.

2.2 Numeric Simplification

The idea of *numerical* simplification is to consider the numerical contribution that a node or subtree makes to the output of its parent node, removing those nodes and subtrees whose impact on the result is too small to make much difference to the program result. For efficiency reasons, this implementation addresses only the local effect of simplification at each node in the program tree. There will be cases where it does affect the system performance of the whole program, but the aim is to keep this to a minimum. It may be easiest to think of numerical simplification as a kind of lossy compression, where we aim to get useful reductions in program size without obvious loss in quality.

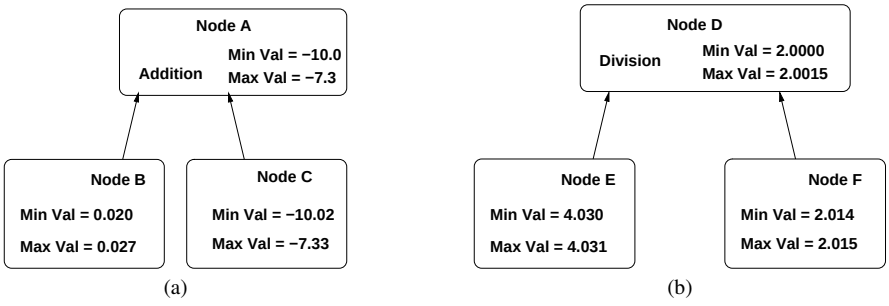


Fig. 2 Examples of numerical simplification.

As the fitness is evaluated across the training set, each node keeps track of minimum and maximum values. The simplification process is performed from the bottom up, so each oper-

ator is responsible for making those simplifications that are meaningful for it. A significance tolerance (threshold) is chosen (set at 0.001 in most of our experiments). For addition and subtraction operators, a child node or subtree whose range of values is less than the threshold times the parent's minimum absolute value is discarded. Figure 2(a) gives an example of such a kind. The range for Node B is $0.027 - 0.020 = 0.007$. The minimum absolute value for its parent Node A is 7.3. Since $0.007 < 0.001 \times 7.3$, the subtree headed by Node B will be discarded, and Node A will be replaced by Node C. Also, if the range of values a node takes is less than the threshold times its own minimum absolute value, the node is replaced by a constant terminal taking its average value. Figure 2(b) gives an example of this kind. The range for Node D is $2.0015 - 2.0000 = 0.0015$. The minimum absolute value for Node D is 2.0000. Since $0.0015 < 0.001 \times 2.0000$, the subtree headed by Node D will be discarded, and Node D will be replaced by a constant terminal with the value 2.00075. Note that the second type of simplification takes precedence over the first.

The numerical simplification method described here has the advantage that it is very simple, both in implimentation and execution, and the process is embedded into the evaluation process. The necessary information is gathered as part of the fitness evaluation and the computational cost is very small. The simplification process then requires one further scan of each program tree, though if simplifications are made then this scan may only be a partial one.

3 Experimental Design

3.1 Experimental Datasets: Three Classification Problems

Fig. 3 Example images from the coin dataset (head and tail).

Coins This dataset consists of a series of 64×64 pixel images of New Zealand five cent pieces against a random noisy background [20]. See Figure 3 for example head and tail images. There are 200 images of each of heads, tails and background only. In this paper, 14 frequency features are extracted based on a discrete cosine transform of the image, as described in [14]. A discrete cosine transform is calculated over the whole image, using an algorithm based on a Fast Fourier Transform for square images where the width is a power of 2. The resulting 64×64 matrix of spectral coefficients contains both frequency and directional information. The directional information is then removed by reducing the matrix to a one dimensional array by averageing each diagonal to give a one dimensional array of 127 non-directional frequency coefficients which are in order from the lowest frequency to the highest. These are then combined into bands to form the features. If C_i is the ith coefficient and $C_{i:j}$ is the average of coefficients i through j, then the features are C_0, C_1, C_2, C_3, C_4, $C_{5:6}$, $C_{7:8}$, $C_{9:10}$, $C_{11:18}$, $C_{19:26}$, $C_{27:34}$, $C_{35:42}$, $C_{43:50}$, $C_{51:126}$.

Wine This dataset [15] gives the result of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantic ties of thirteen constituents found in each of the three types of wines. These thirteen constituents are the features and the three classes are the cultivar from which the wine comes. This dataset was sourced from the Weka project described in [16].

Face Recognition Figure 4 shows three sample images from this dataset. They are from the ORL face data set [12], from which we used the first four individuals thus the set was four classes with ten examples of each. This is rather small and makes evolving a good classifier difficult. As with the coin dataset, the features were based on frequency spectra calculated using a discrete cosine transform.

Fig. 4 Example images from the ORL face dataset.

Eight transforms were used each 16 pixels square. These were placed in various positions within the image. The images are 92 pixels wide and 112 pixels high. The top left corners of the transforms were at pixels [38, 1], [9, 9], [9, 29], [38, 62], [38, 78], [9, 78], [38, 37] and [26, 89]. Six features were created from each transform using the same method as for the coin dataset. The bands were C_0, $C_{1:2}$, $C_{3:4}$, $C_{5:6}$, $C_{7:8}$, $C_{9:30}$. There were therefore 48 features in total. This is a large number given the size of the training set and many of the features are probably redundant. However as the goal of this paper is not to get the best absolute performance but to compare the relative performance between different methods, no effort was made to determine a good subset of features for this problem.

3.2 GP System Configuration

We considered three levels of simplification: *no simplification*, *algebraic simplification*, and *numerical simplification*. The terminal set for each dataset consisted of the features used in that dataset and random "constant" numbers from the range $[0,1]$. All features are normalised over the range $[-1,+1]$. The function set for both datasets consisted of the standard four arithmetic functions and the `IfPos` operator. The `IfPos` operator takes the value of the second child if the first child is greater than zero, and the third child otherwise. The fitness function used the error rate on the training set. The experiments are all conducted with the same set of population parameters. Some preliminary experiments established a set of parameters that gave reasonable classification performance without making too great a demand on memory resources for the no simplification case. The population size was 1000. Initial programs were five levels deep using the full method. Tournament selection was used with a tournament size of four. For the coin dataset we used 40 generations, for the wine dataset we used 200 generations and for the faces dataset 100 generations. Our initial experiments indicated that code growth was influenced by the crossover rate. To find out if this

affected, or was affected by, simplification we applied two sets of parameters for the genetic operators: one 5% reproduction, 85% crossover, 10% mutation; and one 5% reproduction, 15% crossover, 80% mutation. We used ten-fold cross validation, and each combination of dataset, genetic operators and simplification method was run 200 times. On the *numerical simplification* method the *numerical significance threshold* was set at 0.001 for the coin and wine datasets and 0.0005 for the faces dataset. For the various figures that follow, the runs were grouped in to sets of 20, with the results averaged over each of these 20 runs. The graphs intend to show both average behaviour and also compare the variability in those averages. Where simplification was used, it was performed after the first generation, and every fourth generation thereafter. This results in a periodic behaviour in program sizes as will be seen on the graphs that follow. Note that we intend *not* to set any maximum program size as the goal is to investigate whether simplification is sufficient to manage program bloat.

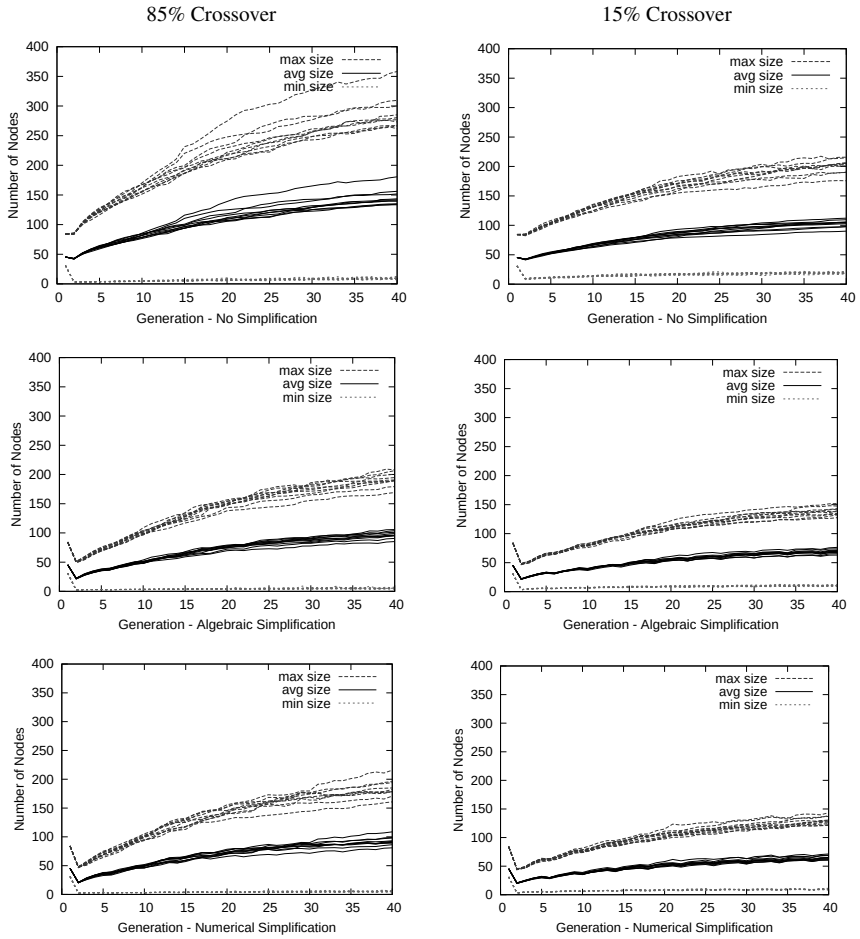


Fig. 5 Program size (number of nodes) for the coin dataset. In each graph, the maximum, mean and minimum program size is shown for 10 replications, each of which is the average of 20 runs.

4 Experimental Results

4.1 Program Size: Number of Nodes

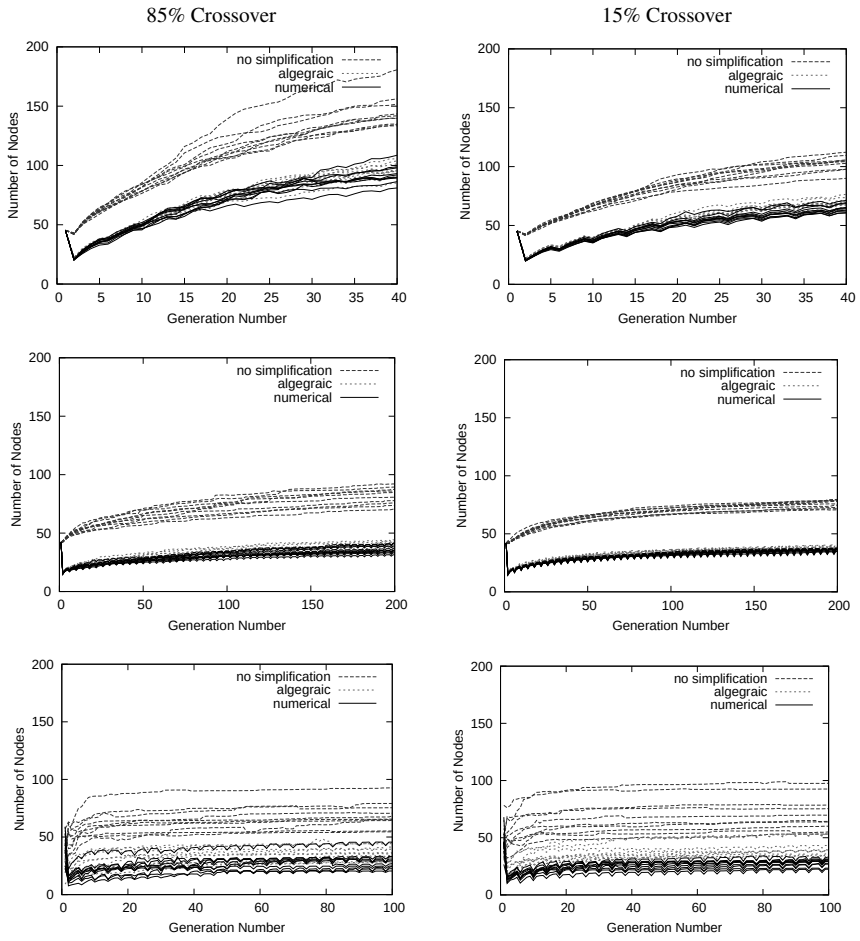


Fig. 6 Mean program size (number of nodes) for the coin dataset (top row), wine dataset (middle row) and the faces dataset (bottom row). For each level of simplification, the mean program size is shown for 10 replications, each of which is the average of 20 runs. In all graphs, the top collection of lines are all no simplification, and the bottom collection of lines are a mixture of algebraic simplification and numerical simplification.

Figure 5 shows the number of nodes per program at each generation for the coin dataset. These graphs show the minimum, mean and maximum program sizes in nodes for each generation in the run. Each curve shows the mean value for that metric across one set of twenty runs. Note that the minimum is small and with little variation. The maximum program size however varies wildly. The lines on these graphs are the mean from 20 runs, individual runs produced numbers from not much larger than the mean up to 3,500 in a single run with 85%

crossover. Larger program sizes are much more common, which pushes up the mean values as well. Program sizes are larger with 85% crossover, and continue to grow even with simplification albeit more slowly than without simplification. Program sizes are smaller with the lower 15% crossover rate, and level off to a steady state with simplification, allowing the possibility of longer runs without the program sizes getting out of control. This difference in performance is likely to be because the replacement sub-trees used by the mutation operator are limited to a maximum depth of five levels and most are three or four levels. In contrast, the replacement sub-tree used by the crossover operator gets larger, on average, as the program becomes deeper. Therefore as the program depth increases a mutation is likely to produce smaller offspring than the crossover operator. In both cases, the program sizes are significantly smaller with simplification than without. The results for the wine and faces datasets show a very similar pattern, so are not shown in detail here.

Figure 6 shows the mean program sizes for all three methods and for all three datasets. With the coin dataset we can clearly see the reduction in program sizes as a result of simplification. While both simplification methods show about a 40% reduction over no simplification, there is no noticeable differences between algebraic and numerical simplification. Finally, we observe the evident periodic nature of the curves, as a result of applying simplification every fourth generation, as described earlier. The results for the wine dataset are very similar to those for the coin dataset. With this dataset there is a small but noticeable advantage to numerical simplification over algebraic for at least the 85% run. The faces dataset shows a lot more variation than the other two datasets but still shows a considerable reduction in program sizes with simplification. There is no clear difference between the two simplification methods in the 85% crossover case but like with the wine dataset the program sizes are noticeably smaller on average with numerical simplification than they are with algebraic simplification when the crossover rate is 15%.

4.2 Program Size: Depth

Figure 7 shows the mean tree depth by generation for the three datasets. It appears that simplification produces slightly shallower trees, but the effect is very small. With the coin dataset, it appears that there is no clear difference between the methods. The wine dataset shows a clearer difference between no simplification and simplification, but no clear difference between the two simplification methods. The results for the faces dataset are very confused. The depth appears to be less in general with simplification than without but it is not possible to draw any more detailed conclusions from this data.

Figures 5 and 7 show a much larger reduction in the number of nodes than in the depth of the trees. This suggests that the reduction in the number of nodes in the tree was primarily due to thinning of the tree (reducing the number of nodes at each level) rather than reducing the depth of the tree. An additional series of experiments was performed with extra data collection to verify this. There were 50 runs for each method and only the coin dataset was examined. Figure 8 shows the resulting average number of nodes at each level(depth). The four graphs show the data at four different generations: 5, 15, 25 and 35. The total number of nodes in the population is proportional to the area under the curves so we are interested in how the shape of the curves change when we use simplification. We can see the curves broaden as the run proceeds through the generations because of the increase in the average program depth; this is also true for the runs with simplification. We can also see that the effect of simplification at all generations is to reduce the height of the curve, rather than to narrow it or to move the peak towards a lower level. This shows that simplification

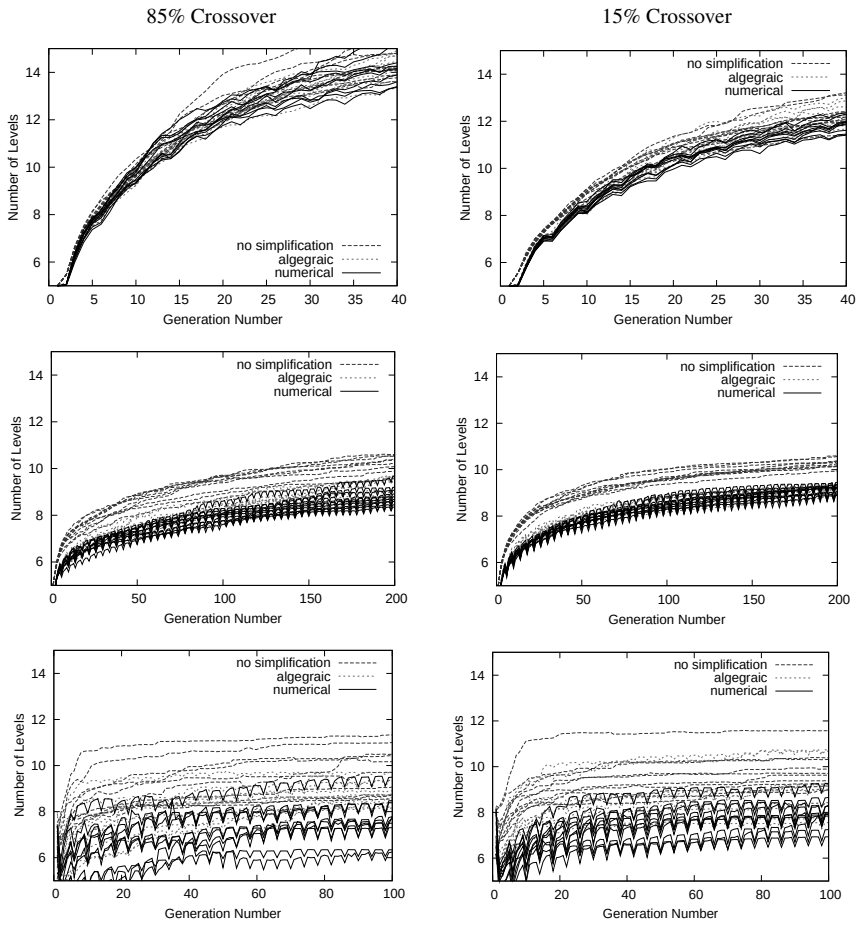


Fig. 7 Mean tree depth for the coin dataset (top row), wine dataset (middle row) and the faces dataset (bottom row). Again these are 10 replications (curves), each of which is the average of 20 runs.

is reducing the average number of nodes at the middle levels of the tree, rather than by reducing the depth of the tree.

4.3 Analysis of Resource Usage

Figure 9 shows boxplots for memory use and CPU time. There is a pattern here, with simplification giving a noticeable reduction in CPU usage. Algebraic simplification shows a reduction in run times of 19–40% compared to no simplification and numerical simplification 25–54%. Overall, numerical simplification run times averaged 6% shorter than algebraic simplification. There is also a reduction in memory usage, with simplification showing a reduction in memory usage of 30–40% in most cases, and a small advantage of numerical simplification over algebraic simplification.

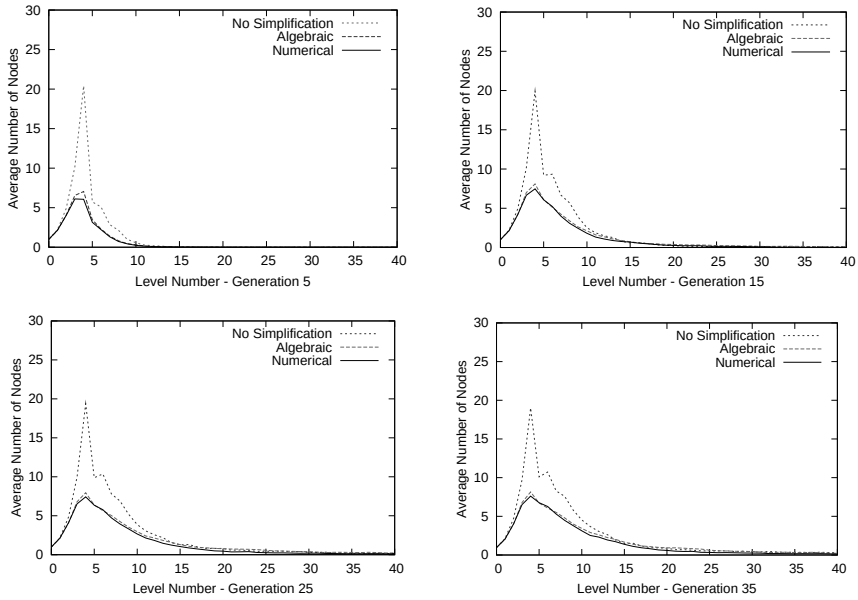


Fig. 8 The average number of nodes at each level at four different generations.

4.3.1 Significance of differences in CPU resource usage

The work described above shows the basic behaviour of the two simplification methods on three datasets. It indicates that there are differences between them in program sizes and resource usage but does not give any indication of significance. To investigate this aspect, we look at the differences between *no simplification*, *algebraic simplification*, and *numerical simplification*. We maximise the corelation between the starting points for the different simplification approaches. Therefore each set of three experiments (one for each of *no simplification*, *algebraic simplification* and *numerical simplification*), used the same initial population, and the same split of the training set into folds. Each set of three experiments had a different initial population but retained the same training set split. For the coin and wine datasets there were 200 such runs, each one a ten-fold cross validation, for the faces dataset there were 50 runs.

Figure 10 shows box plots for the *differences* in CPU time for the different datasets and simplification methods. There is a clear advantage to the two simplification methods with all three datasets. The advantage is both greater, and with better significance on the wine and faces datasets, particularly the latter. On the coins and wine datasets there is no significant difference between algebraic and numerical simplification, but on the faces dataset numerical simplification shows a significant reduction in CPU time compared to algebraic simplification.

4.3.2 Statistical Significance Scores

The distribution of differences in Figure 10 is not a normal distribution as there are long tails, and in many cases the distribution is badly skewed. This rules out the use of the standard *student T*–test. Instead we use a non-parametric test called the Wilcoxon Signed-Rank

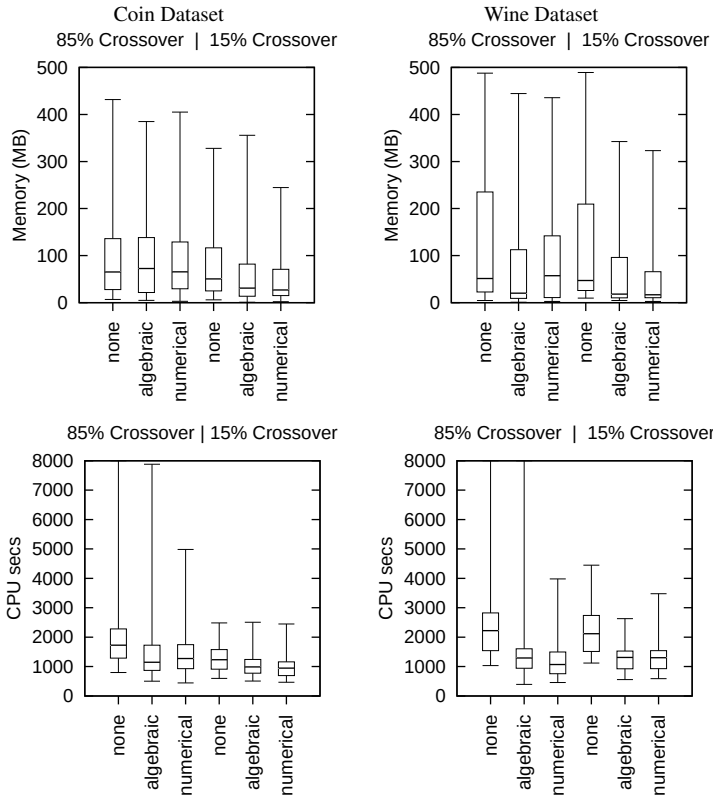


Fig. 9 CPU and memory usage for the coin dataset (left) and wine dataset (right) for all 200 runs. The boxplots on each graph are (left to right) 85% crossover with no simplification, algebraic simplification, and numerical simplification, then 15% crossover with no simplification, algebraic simplification and numerical simplification.

test [17, 18]. This does not require that the distributions are normal and it uses only the order of the results, not the magnitude. It does require the distributions to be symmetrical, this requirement can be met by testing the median rather than the mean of the differences distributions.

Table 2 shows the median values for the CPU time used by the various experiments. For the difference lines, the third column gives the Z score calculated using the Wilcoxon Signed-Rank test. Table 3 shows the critical Z values. Note that these are for a directional test. That is the displayed Z values give the confidence that the simplification method uses less CPU than *no simplification* or that *numerical simplification* uses less CPU than *algebraic simplification*. Where the Z score warrents it, the fourth and fifth columns show the 95% confidence interval expressed as the percentage reduction in CPU time used.

Each set of four lines gives the following:

1. CPU time used by the *no simplification* runs.
2. The reduction in CPU used by the *algebraic simplification* runs compared to the *no simplification* runs.

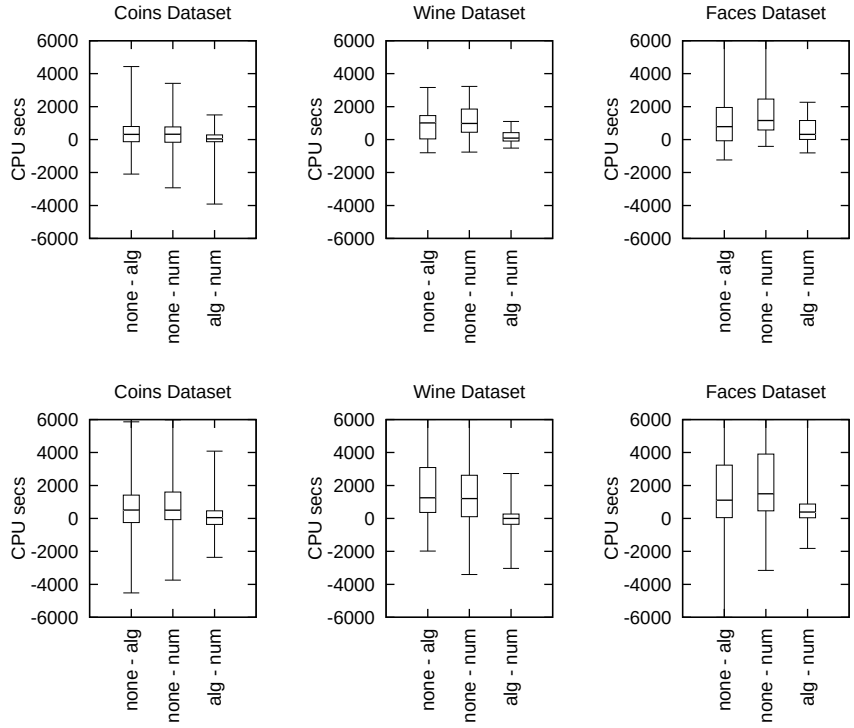


Fig. 10 Differences in CPU resource used for the three datasets. The top row is for 15% crossover and the bottom row is for 85% crossover. In all graphs the first box is the difference between no simplification and algebraic simplification, the second is the difference between no simplification and numerical simplification and the third is the difference between algebraic and numerical simplification.

3. The reduction in CPU used by the *numerical simplification* runs compared to the *no simplification* runs.
4. The reduction in CPU used by the *numerical simplification* runs compared to the *algebraic simplification* runs.

We can see that for *numerical simplification* and 15% crossover on the coin data set the Z score is just under the 95% confidence level but the confidence interval still looks good. All the other differences between *simplification* and *no simplification* are significant to at least 99% confidence. The differences between the two simplification methods show no meaningful significance for the coin dataset, and for 85% crossover on the wine dataset. The differences between them is however highly significant for 15% crossover on the wine dataset and for both cases on the faces dataset. When we introduced the faces dataset we noted the fairly large number of features, and one reason for such a large reduction in program sizes and CPU time with *numerical simplification* with this dataset might be that some implicit feature selection is occurring. Features that have little effect on fitness may be being discarded during simplification.

	Median	Z	Min %	Max %
Coins 15% Crossover with no simplification	1622			
Reduction with Algebraic simplification	518	**2.58**	28	40
Reduction with Numeric simplification	444	1.59	17	30
Difference between Algebraic and Numeric Simplifications	34	0.15		
Coins 85% Crossover with no simplification	2420			
Reduction with Algebraic simplification	468	**2.49**	24	34
Reduction with Numeric simplification	798	**2.64**	26	37
Difference between Algebraic and Numeric Simplifications	11	0.03		
Wine 15% Crossover with no simplification	2390			
Reduction with Algebraic simplification	1009	**4.75**	35	40
Reduction with Numeric simplification	976	**5.51**	43	46
Difference between Algebraic and Numeric Simplifications	90	**2.15**	4	6
Wine 85% Crossover with no simplification	2116			
Reduction with Algebraic simplification	1254	**5.07**	63	75
Reduction with Numeric simplification	1207	**4.25**	55	66
Difference between Algebraic and Numeric Simplifications	−1	0.82		
Faces 15% Crossover with no simplification	2374			
Reduction with Algebraic simplification	782	**4.03**	40	51
Reduction with Numeric simplification	1160	**5.93**	57	67
Difference between Algebraic and Numeric Simplifications	315	**4.51**	17	21
Faces 85% Crossover with no simplification	2808			
Reduction with Algebraic simplification	1104	**3.01**	40	51
Reduction with Numeric simplification	1495	**4.84**	61	71
Difference between Algebraic and Numeric Simplifications	390	**3.69**	13	16

Table 2 CPU time used in seconds for the coins, wine and faces datasets.

4.4 Analysis of Classification Performance

Reducing the program size and computational effort is of little benefit if the classification performance suffers badly. Figure 11 gives the mean classification error rate on the test set for the program with the best fitness at each generation for the coin, wine and faces datasets, again showing 10 replications, each of which is the average of 20 runs. There is little noticeable difference in performance between the three levels of simplification on the coin dataset. The wine dataset shows similar results but with more variation in classification performance. The average performance for the wine dataset is a classification error rate of between 15% and 20%, but some individual runs produced classifiers with zero error rate. The classification performance also appears to be sensitive to which examples are included the test set. The results for the faces dataset show a much wider variation in classification performance, but there is no apparent overall difference between the three methods.

Significance Level	0.20	0.10	0.05	0.025	0.01	0.005	0.0005
Z value	0.842	1.282	1.645	1.960	2.326	2.576	3.291

Table 3 Critical Z values

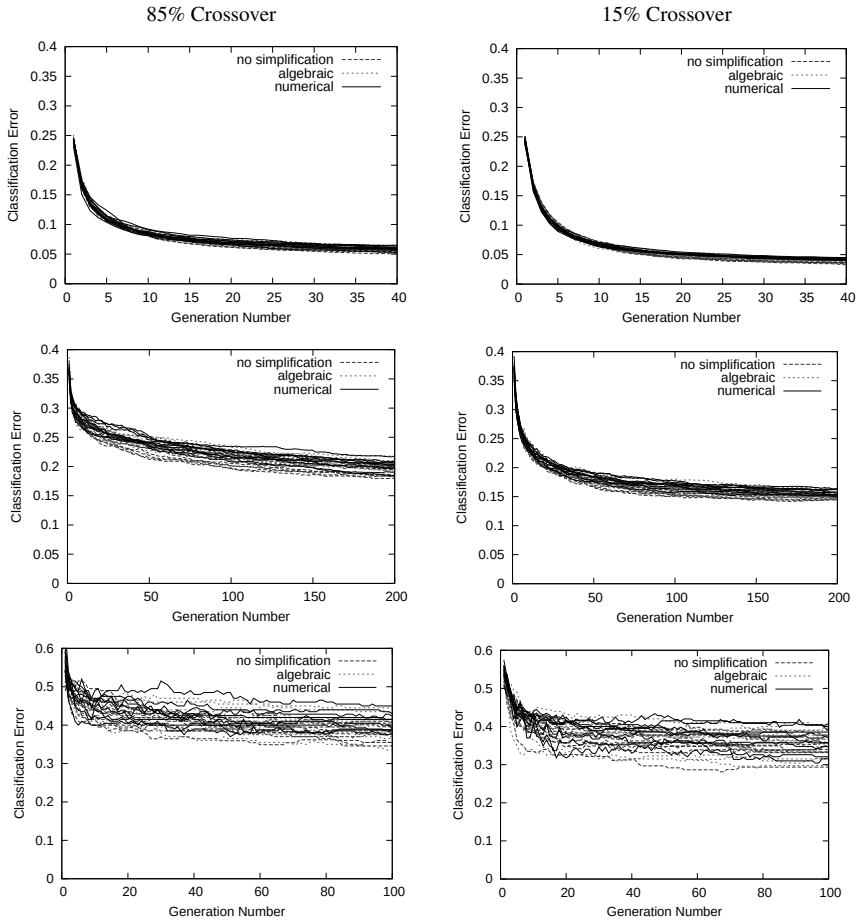


Fig. 11 Classification performance for the coin dataset (top row), the wine dataset (middle row) and the faces dataset (bottom row).

Note that with all three datasets, and both simplification methods, that the 15% crossover case produces both smaller programs and lower classification error rates on the test set than the 85% case. This may be because of the smaller program size, or some other effect of the difference in genetic operators. More targeted experiments will be needed to establish this.

Overall the two simplification methods always resulted in much smaller programs and used much shorter evolutionary training times to produce similar classification performance. *numerical simplification* performs at least as well as *algebraic simplification* and in some cases performs better than *algebraic simplification*.

5 The Effect of Simplification on Building Blocks

5.1 Encoding the Building Blocks

Wong and Zhang [19] examined the effect of *algebraic simplification* on building blocks by tracking the ephemeral constants in the population. In standard GP these are constant for the duration of the run, but both simplification approaches can change the value of such constants and also create new ones. They showed that while algebraic simplification does disrupt building blocks by destroying or changing constants, it is also capable of creating values that were not originally present and that some of these new values became established in the population thus contributing to the final solution.

What we wish to examine is the behaviour for larger building blocks, in particular two and three level deep sub-trees. Like [19] we will present this behaviour as images. In any practical problem, the number of possible building blocks is huge, far larger than we can attempt to enumerate let alone display. We need a way of showing at least the nature of the building blocks present in the population, and an indication of relative frequency. This is a very difficult problem, particularly as the size of the building blocks increase. Our approach is to encode each building block into a bit string in such a way that similar building blocks result in similar encodings. The encoding process simplifies the description of the nodes to keep the size of the bit string manageable. In each case the nodes are encoded one level at a time, starting from the root, and from left to right within the level. Figure 12 shows a three level deep subtree. In this example the order the nodes would be encoded is $\times$ $-$ $+$ $-$ $F2$ 0.3 $F3$.

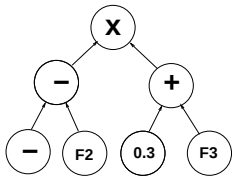


Fig. 12 An example tree to illustrate the encoding order.

We wish to show building blocks as part of the whole search space. This is so we can see what coverage we are getting and how this might change with simplification. The challenge is in displaying this information. The chosen format is an image, with building blocks on the vertical axis and generations on the horizontal. Even at 1200dpi there is a very limited number of pixels and therefore bits available for encoding the building blocks if the resulting image is to be less than a page in size. The goal is to handle trees up to a depth of three levels.

A building block three levels deep, with all operators having two arguments, has either five or seven nodes. An image 6.83 inches high at 1200dpi gives us 13 bits, or 14 bits at 2400dpi. That would allow only two bits per node. The encoding scheme can therefore give only the most general indication of the structure of the building block. If the building block is only two levels deep, then we could use two bits for the operator at the root, and five bits for each of the two child nodes. The two encoding schemes we chose are described in the rest of this section.

5.1.1 Three Level Deep Building Blocks.

With two bits per node there are four available encodings for each node. The ones we have used are:

1. 00 - *addition* or *subtraction* operator.
2. 01 - *multiplication* or *division* operator.
3. 10 - input feature.
4. 11 - ephemeral constant.

Because the root node of a building block at least two levels deep must be an operator, the first bit will always be zero leaving us with 13 bits describing a three deep building block. $2^{13} = 8192$ which at 1200dpi gives a graphic 6.83 inches high which is feasible for printing.

As described above, the encoding order for Figure 12 is $[\times]$ $[-]$ $[+]$ $[-]$ $[F2]$ [0.3] $[F3]$, which is encoded as [1] [00] [00] [00] [10] [11] [10], the resulting encoding is then 1000000101110.

At each generation we traverse all programs in the population, encoding all building blocks that are three levels deep. A hash table is used to record all of the encodings found and to accumulate the number of times they occur.

At the end of the run a graphic is created with a height of 8192 pixels, one for each possible encoding and the width being the number of generations. All non-zero counts are normalised on to a range 64–255 to make a clear visual greyscale distinction between a low count and zero. The final image is then produced from the negative, so the highest count is black with zero being white.

5.1.2 Two Level Deep Building Blocks.

This time we have more bits available to describe each node. The root node is always an operator and uses two bits with the values:

1. 00 - *addition* operator.
2. 01 - *subtraction* operator.
3. 10 - *multiplication* operator.
4. 11 - *division* operator.

The two child nodes each have five bits, used as follows:

1. 1 followed by four bits is a feature, with the four bits being the feature number.
2. 01 followed by three bits is an ephemeral constant with the absolute value used to map [0.0,1.0] on to the [0,7] range allowed by the three available bits.
3. 000 followed by two bits is an operator, enumerated as for the root node.

This scheme allows more information about the building block to be included in the encoding than was the case for the three level deep building blocks and uses a total of twelve bits to describe each building block. $2^{12} = 4096$ which at 1200dpi makes our graphic 3.42 inches high.

In the example of Figure 13, the order of encoding is $[/]$ [0.3] $[F3]$ where F3 is input feature number three. This then encoded as [11] [01 010] [1 0011] giving an encoding of 110101010011. The rest of the processing is as for the three level case except the graphic is now only 4096 pixels in height.

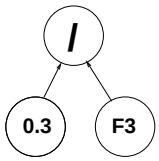


Fig. 13 An example tree to illustrate the encoding scheme for two deep building blocks.

5.2 Analysis of Building Blocks

The particular runs and their associated images that we present here are representative of the behaviour we observed across many other similar runs. In addition, this section only shows the analysis on the coin and wine datasets as the analysis on the face dataset gave a very similar pattern. The images have been stretched in width to make them easier to see, so each generation for the coins dataset is 10 pixels wide, and for the wine dataset 2 pixels wide. This gives a common size of image. Similarly the images for the two level building blocks have been scaled vertically to match those for three level building blocks.

A very important property of both the encoding schemes used to create these images is that pixels in the image that are close together vertically, in general represent closely related building blocks. For the three deep encoding the differences will generally only be in the third level. With the two deep level encoding the differences will in most cases be a different value for an ephemeral constant, or a different feature being used.

5.2.1 Three Level Building Blocks

Figure 14 shows three images/plots for the coin dataset, one example run of each GP system (*no simplification*, *algebraic simplification* and *numerical simplification*). Recall that the vertical axis represents each of the possible 8192 building blocks, the horizontal axis represents the possible generations during evolution, and that all the three runs start with the same random seed and the same initial population so that they have the same starting point. An enlarged view of a densely populated area and a sparsely populated area is shown in Figure 16. In these figures, continuous solid lines represent good building blocks retained during evolution, while discontinuous "lines" (or scattered dots/short lines) indicate building blocks were disrupted. The darkness of the lines/dots represents the occurrence frequency of the building blocks.

Inspection of Figure 14(a) reveals that in the canonical GP system with *no simplification*, many useful building blocks are retained in the entire evolutionary process (some shown clearly in areas C and D and some others), while many other building blocks were disrupted (mainly by crossover and some by mutation) during evolution. In addition, many of the 8192 possible three-level subtrees that were present in the initial population died during evolution.

Inspection of Figure 14(b) and (c) reveals that there are still many good building blocks (dark continuous solid lines) during evolution for two different simplification systems. Although some of the good building blocks originally retained during evolution in the canonical GP system with no simplification were destroyed (lines were broken), some new building blocks were produced by the simplification process during evolution.

Figure 15 shows the plots on the wine dataset for the three systems (*no simplification*, *algebraic simplification* and *numerical simplification*). While the building blocks behave differently in these images from those in Figure 14, the main pattern remains the same: the two simplification processes do destroy existing building blocks but they also generate new

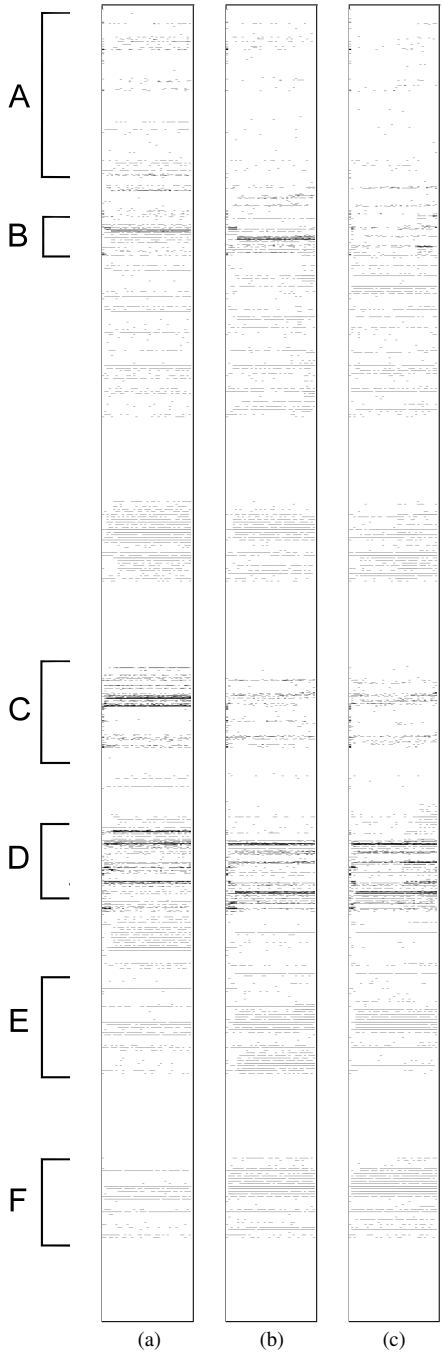


Fig. 14 A set of runs for the coins dataset with three deep building blocks. (a) with no simplification. (b) with algebraic simplification. (c) with numerical simplification.

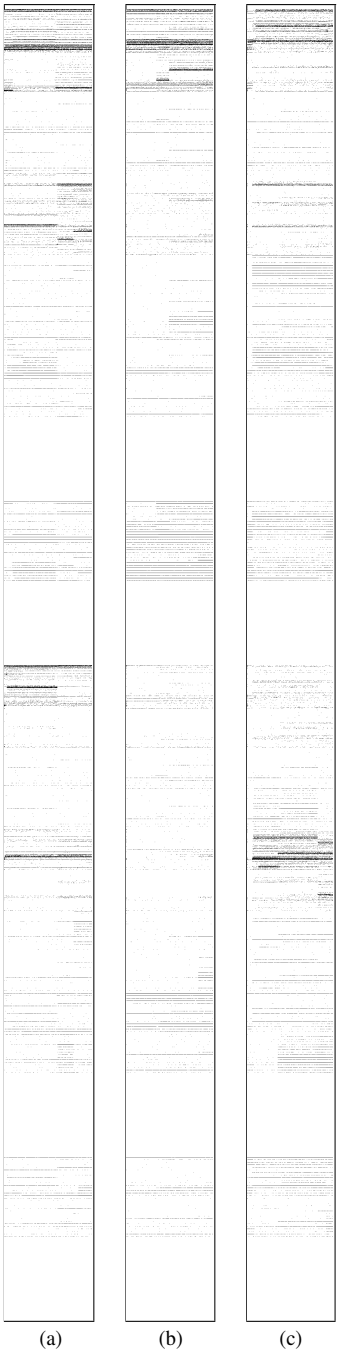
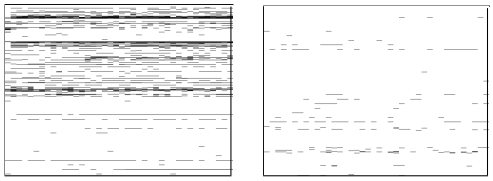


Fig. 15 A set of runs for the wine dataset with three deep building blocks. (a) with no simplification. (b) with algebraic simplification. (c) with numerical simplification.



(a) (b)

Fig. 16 Enlargements of part of an image for the coins dataset with no simplification. (a) is from a densely populated area and (b) is from a sparsely populated area.

and more diverse building blocks during evolution. These new building blocks contribute to problem solving, so the classification performance was retained.

Comparing many runs on the two datasets, we found that fewer existing building blocks were retained and more new building blocks were generated for both simplification methods in the wine data set than in the coins dataset. This is due mainly to the degree of difficulty in the classification problems in the two datasets. The classification problem in the wine dataset is more difficult than in the coins dataset.

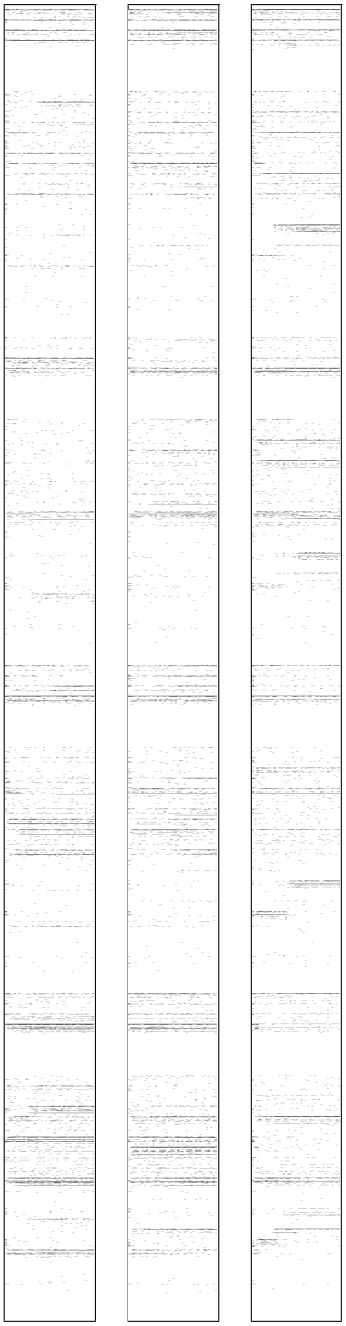
5.2.2 Two Level Building Blocks

The three level deep building blocks do not distinguish between addition and subtraction or between multiplication and division. To make this distinction, we examine the behaviour of the two-level deep building blocks to check whether the conclusions still remain valid.

Figure 17 and Figure 18 show the plots for two level building blocks for the coin and wine datasets respectively using GP with *no simplification*, *algebraic simplification* and *numerical simplification*. These results show a very similar pattern to those on the three-level building blocks in the previous subsection.

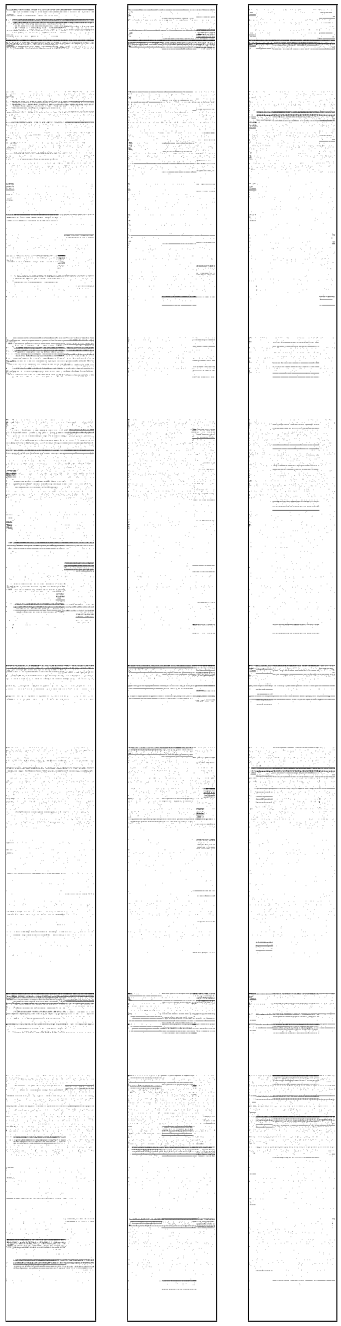
While the building blocks of two and three level subtrees behave differently from the numerical "constant" terminals only in [19], the main conclusions remain the same: the two program simplification algorithms destroy existing building blocks, generate new and more diverse building blocks, and the contributions of the new building blocks compensate the negative aspects of the existing building block disruption.

One observation from these new results is that many useful existing building blocks are preserved during evolution. Wong and Zhang's analysis on the simplest building blocks (numeric/constant terminals) [19] did not clearly show this. This is perhaps because the format of the numeric terminals is a single floating point number and is too simple to show this pattern.



(a) (b) (c)

Fig. 17 A set of runs for the coins dataset with two deep building blocks. (a) with no simplification. (b) with algebraic simplification. (c) with numerical simplification.



(a) (b) (c)

Fig. 18 A set of runs for the wine dataset with two deep building blocks. (a) with no simplification. (b) with algebraic simplification. (c) with numerical simplification.

6 Conclusions

We have shown that both simplification methods appear to reduce the average number of nodes per program by about 40%. Some of this is by reducing the depth of the tree, but most of the reduction is by reducing the average number of nodes per level in the upper and middle parts of the tree.

We have also shown that simplification reduces CPU usage by between 18% and 74% on these experiments. The direct measurements of memory usage show a reduction of about 30% but measurements vary widely due to difficulties with measuring memory use within a Java runtime environment. In practice, most of the memory used is taken up by the program population.

Numerical simplification is at least as effective as algebraic simplification, while showing some advantage in resource usage in many but not all cases. This depends on the dataset, with the face and wine datasets showing an advantage to numerical simplification in program size and resource usage. With the coin dataset there was no significant difference observed.

We have shown that there appears to be no significant loss in classification performance, provided that the numerical simplification threshold is set appropriately.

We introduced encoding schemes for two and three level deep building blocks that have shown the distribution of building blocks within a population using images. These revealed that although the two online simplification methods destroyed some existing building blocks, they effectively generated additional new and more diverse building blocks during evolution, which compensated for the negative effect from the disruption of building blocks. These findings further confirmed the early hypothesis and results made by Wong and Zhang [19], where the analysis was based only on the simplest form of building blocks, numerical constants, on two simple regression tasks. This in turn concludes that the two online program simplification methods can produce significantly smaller programs and take significantly shorter evolutionary training time than the canonical GP system while achieving comparable effectiveness performance.

One interesting observation from this paper is that many existing building blocks were preserved during evolution, although many building blocks change over time with generations. In the future, we will make further analysis to determine why so many existing building blocks were preserved, and which building blocks they actually are. This will help in understanding the internal behaviour of the two online simplification methods, which in turn will reveal the differences between them.

The images presented here are based on many individual GP runs. It would be very interesting to further investigate the distribution of the building blocks over multiple combined runs (i.e. 100 or 200 runs) in the future to reveal whether the findings obtained here will still remain valid.

Acknowledgement

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

References

1. 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, MIT Press (1996) 215–223
2. Soule, T., Heckendorn, R.B.: An analysis of the causes of code growth in genetic programming. Genetic Programming and Evolvable Machines **3**(3) (2002) 283–309
3. Blickle, T., Thiele, L.: Genetic programming and redundancy. In Hopf, J., ed.: Genetic Algorithms within the Framework of Evolutionary Computation. (1994) 33–38
4. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications. (1998) Morgan Kaufmann Publishers.
5. Zhang, M., Smart, W.: Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. Pattern Recognition Letters **27**(11) (2006) 1266–1274
6. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
7. Nordin, P., Banzhaf, W.: Complexity compression and evolution. In Eshelman, L., ed.: Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), Pittsburgh, PA, USA, Morgan Kaufmann (15–19 July 1995) 310–317
8. Parrott, D., Li, X., Ciesielski, V.: Multi-objective techniques in genetic programming for evolving classifiers. In Corne, D., Michalewicz, Z. et al., eds.: Proceedings of the 2005 IEEE Congress on Evolutionary Computation. Volume 2., Edinburgh, UK, IEEE Press (2–5 September 2005) 1141–1148
9. Zhang, B.T., Mühlenbein, H.: Balancing accuracy and parsimony in genetic programming. Evolutionary Computation **3**(1) (1995) 17–38
10. Zhang, M., Bhowan, U.: Program size and pixel statistics in genetic programming for object detection. In Raidl, G.R., Cagnoni, S., et al. eds.: Applications of Evolutionary Computing, EvoWorkshops2004. Volume 3005 of LNCS., Coimbra, Portugal, Springer Verlag (5–7 April 2004) 379–388
11. Luke, S., Panait, L.: Lexicographic parsimony pressure. In Langdon, W.B., et al., eds.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, Morgan Kaufmann Publishers (9–13 July 2002) 829–836
12. Samaria, F. and Harter, A. C.: Parameterisation of a stochastic model for human face identification. Proceedings of the Second IEEE Workshop on Applications of Computer Vision (1994)
13. de Jong, E.D., Pollack, J.B.: Multi-objective methods for tree size control. Genetic Programming and Evolvable Machines **4**(3) (2003) 211–233
14. Marshall, D.: The discrete cosine transform. (2001) http://www.cs.cf.ac.uk/Dave/Multimedia/node231.html
15. Forina, M., Leardi, R., Armanino, C., Lanteri, S.: Parvus: an Extendable Package of Programs for Data Exploration, Classification and Correlation. Elsevier, Amsterdam (1988)
16. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2nd ed. Morgan Kaufmann, San Francisco (2005)
17. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics, 1, 80-83. (1945).
18. LaVange, Lisa M., Koch, Gary G.: Rank Score Tests. In Circulation. Volume 114. Number 23, 2528–2533 (2006)
19. P. Wong and M. Zhang. Effects of program simplification on simple building blocks in genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 1570–1577, 2007.
20. P. Wong and M. Zhang. Algebraic simplification of GP programs during evolution. In M. Keijzer, et al. editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and Evolutionary Computation*, volume 1, pages 927–934, USA, 2006. ACM Press.