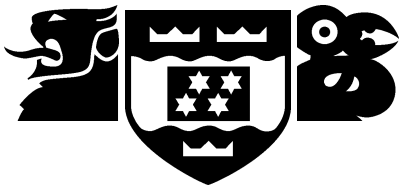
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

A Multiple-Output Program Tree Structure in Genetic Programming

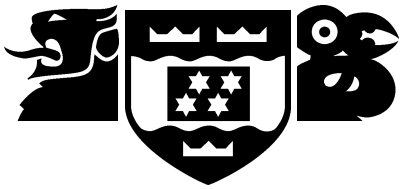
Yun Zhang, Mengjie Zhang

Technical Report CS-TR-04/14
December 2004

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

A Multiple-Output Program Tree Structure in Genetic Programming

Yun Zhang, Mengjie Zhang

Technical Report CS-TR-04/14
December 2004

Abstract

This paper presents a new program tree structure in genetic programming which outputs multiple related values, hence serves as a more coherent multiclass classifier. The multiple outputting effect of the tree is achieved by making it simulate a kind of directed acyclic graph. The approach is examined and compared with the basic genetic programming approach on four multiclass object classification tasks with varying difficulty. The results show that the new approach greatly outperforms the basic genetic programming approach on all the tasks.

Keywords Genetic programming, image recognition, multiple class object classification, modifying-based program structure, Modi.

Author Information

Yun Zhang is a postgraduate student in computer science and Mengjie Zhang is an academic staff member in computer science. Both authors are in the School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand.

1 Introduction

Genetic programming (GP) is an optimization technique that learns by following a genetic beam search on a set of candidate solutions (Koza, 1992). Different representations of the solution result in different families of GP. *Linear GP* manipulates on solutions in the form of sequence of imperative instructions (C, machine code), whereas *tree-based GP* manipulates on tree structured programs (Lisp). It has been shown that the two different representations have their own strength and weakness.

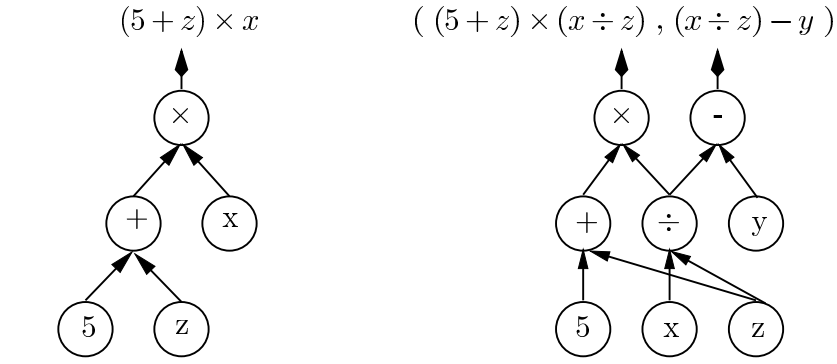
This paper focuses on the tree-based GP, in which solutions are in the form of program trees that are similar with the one shown in Figure 1, though often much bigger. When being evaluated, the program tree takes inputs from leaf nodes, applies functions on them feedforwardly, finally releases a single output from the tree root. In the standard tree-based GP presented in (Koza, 1992), all program trees and subtrees have the same data type to ensure that free crossover and mutation yield valid trees. In principle, this "typeless" constraint should not reduce the computational power of the genetic program tree, as the system is still Turing complete. However, many application problems, particularly those involving matrices and lists, are hard to be represented without types (Hopper and Reierson).

In other words, under the closure property, the genetic program tree can only work as a many-to-one function over an identical datatype. This will cause problems when the tasks naturally desire many-to-many solutions. A typical example of this case is *multiclass classification*, whose solution is in the form of a classifier that could determine which class a particular example belongs to over multiple classes based on some input values. With multiple outputs, one can consider each of the outputs corresponding to a distinct class. Multi-layer feed forward neural networks, the current dominator of this area, usually uses exactly this approach (Rumelhart et al., 1986).

The research of applying the standard tree based GP on multiclass classification is an active area. Previous works are mainly focused on how to transform the single output value of the program into a class label over multiple classes (Loveard and Ciesielski, 2001; Zhang and Smart, 2004). Solutions are mostly to partition the range of the program output into regions, one per class, then consider the class corresponding to the region that the output value lies in as the result. The range partitioning, which needs to decide both region boundaries and the class ordering, can either be manually predefined, which would require domain knowledge and/or expensive trials, or automatically learnt, which would be time consuming and often result in unnecessarily complex programs.

The undesirable range partitioning problem could be eliminated with a multiple-output solution representation of genetic programs. Along this track, one old idea was to use a sequence of independent programs, one regarding each output, to gain a multiple-output effect. However the independency over outputs does not reflect the relationship/relavance between different classes.

Another idea was to use the *multiple roots tree* (MRtree) structure, as shown in Figure 1. This structure is actually the *loppy directed acyclic graph* (LDAG), with nodes of no in-edges considered leaves, and nodes of no out-edges considered roots. "Loppy" means that we do desire cycles if one ignores the directions of arrows, which will overcome the problem of using a sequence of program trees, as addressed above. However, evolving this structure by the standard tree-based GP is difficult. Firstly, the initial solution space of MRtree is hard to generate. One needs to consider the reuse of nodes and the acyclic property. Secondly, applying tree-based crossover and mutation to MRTrees would yield invalid child programs.



Standard Program Tree (SDtree)

Multiple Roots Program Tree (MRtree)

This paper introduces a new structure called *Modi*. Its name refers to the idea of making the program tree work as a ***modi****fying* component that computes outputs by modifying a predefined output structure, in contrast to the traditional *outputting* based program tree structure, which releases the output from the root node of the program tree. The elite of Modi is that, it is structurally equivalent to the standard program trees (SDTree) thus preserves the consistency of crossover and mutation, yet it is functionally equivalent to the MRTree thus reasonably outputs a list of values.

The rest of the paper is organized as follows. Section 2 describes the new program structure, Modi. Section 3 describes the experiment configuration and image data sets. Section 4 presents the results with discussions. Section 5 draws the main conclusions and gives future working directions.

2 The Modi Program Tree Structure

2.1 Modi Program Structure

The Modi program structure has two main parts: (a) a program tree, and (b) an associated *output vector* for holding outputs, as shown in Figure 1.

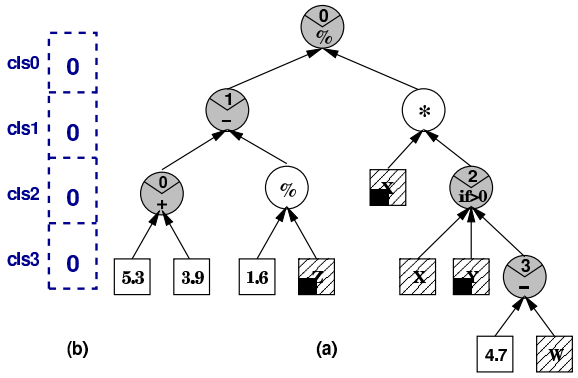


Figure 1: An example Modi program structure. (a) Modi program tree; (b) Output vector.

Similarly to the standard program tree structure, the Modi tree also has a root node, internal nodes and leaf nodes. Feature terminals (slashed squares) and random constants (clear squares) form the leaf nodes. Operations in the function set form the root and internal nodes (circles).

Unlike the standard program tree structure, which outputs just one value (often a floating point number) through the root, our Modi program structure takes the *output vector* as the

output space, hence produces multiple values, each of which corresponds to a single class in the multiclass classification problem.

The two parts of the Modi structure, the output vector and the Modi program tree, are not structurally connected but connected through some special function nodes, *Modi nodes* (grey circles). A Modi node has two roles: (1) It *updates* an element in the output vector that the node is pre-associated with, by adding its node value to the value of the vector element; (2) It passes the value of its right child node to its parent, so that the program tree can be continuously preserved.

Note that the output vector is considered *virtual* (that is why it is dashed in the figure), meaning that it does not physically "exist" (i.e. take up memory) other than the moment when the program is being evaluated. This means, during GP's evolutionary learning, output vectors of all programs "disappear" and only Modi program trees are active. Only in the program evaluation time, is the output vector realized and receives updating from the program tree.

2.2 Evaluating the Modi Program

Figure 2 shows what happens while the example program is evaluated. Before the evaluation starts, the virtual output vector is initialized with zeros. During the evaluation, each *non*-Modi node passes its value to its parent, exactly the same as in the standard program tree. Modi nodes work differently. Each Modi node firstly uses its node value to update the output vector (shown as curved solid arrows), instead of passing the node value to its parent node as in the standard program tree (that is why the connection between modi nodes and their parents are shown in fine un-arrowed lines). Modi nodes then passes on the value of its right child to its parent node (shown as dashed arrows). The consequence of the program evaluation is that the output vector gets properly updated by Modi nodes in the program tree. Multiple floating point numbers are then produced, each of which corresponds to a class. Finally, a voting strategy is applied to those outputs. The winner class (the one with the maximum value) is considered to be the class of the input pattern.

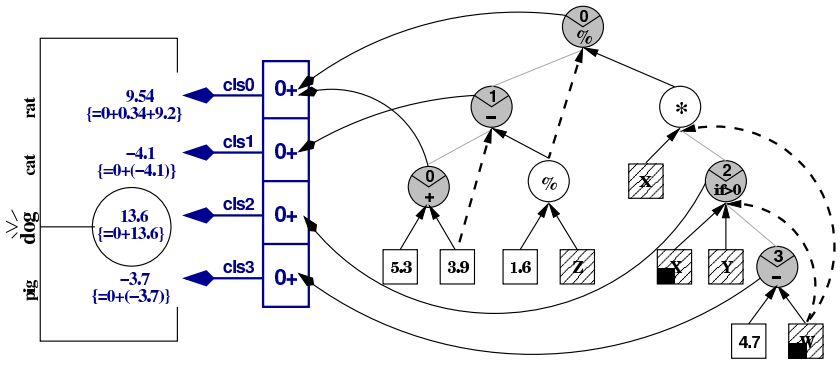


Figure 2: Evaluation of the example Modi program structure.

Consider a four class classification task with possible classes {`rat,cat,dog,pig`} and an object to be classified – *doggie*, represented by an input vector that consists of six extracted feature values `[V,U,W,X,Y,Z]=[0.6,5.7,8.4,2.8,13.6,0.2]`. Assuming the Modi program shown in Figure 1 is the learnt classifier, feeding the input vector into the classifier would *modify* the output vector to `[9.54,-4.1,13.6,-3.7]`, as shown in the left part of figure 2. Given this result, *doggie* is classified into the third class `dog`, as the third output `13.6` is the *winner*.

2.3 The DAG (MRtree) Simulation Effect of Modi

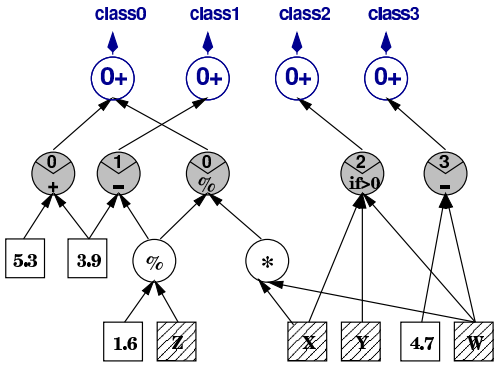


Figure 3: Modi simulated graph/network classifier.

Figure 3 is just a tidy-up redrawing of Figure 2 with the structural-used-only fine grey line removed. The figure clearly shows that the dynamic running effect of the Modi program actually simulates the MRTree that is a loppy *Directed Acyclic Graph*, although the real structure of Modi is just a normal tree plus a vector. Similarly to the multi-layer feed forward neural networks, which are also a kind of DAG, the Modi simulated DAG also has multiple layers, where leaf nodes form the input space, internal nodes extract higher level features, and output nodes are associated with class labels. However, it allows unbalanced structure, over-bipartite connections, and non-full connections between neighboring layers, which makes the representation much more flexible.

By the effect of *loppy*, the simulated MRtree structure also allows the *reuse* of child nodes. Every right most child of the Modi node is reused by the Modi node itself and the parent of the Modi node, resulting in a two-way reuse. Multi-way reuse is also possible by a sequence of hierarchically connected Modi nodes, as shown at the bottom right corner around the feature terminal node W.

2.4 Modi Program Generation and Modi Rate μ

Compared with SDTree, the initial generation process of Modi trees has one more step to consider, namely the distribution of Modi nodes. This includes two points:

point1: How to choose Modi nodes, and

point2: How to assign output vector's cell indices to the Modi nodes we would have chosen.

The solution of the first point consists of three rules: 1) All leaf nodes are not Modi since a Modi node requires at least one child; 2) The root node is always Modi, so that we can guarantee that no part of the Modi tree is useless, as we do not make explicit use of the value released from the tree root; 3) For intermediate nodes, the probability of a node to be set to Modi is defined by an offline settable constant μ, the *Modi Rate*. Modi Rate is a predefined setting of the GP evolutionary learning system. It is in the form of a real number $\mu \in [0, 1]$, which refers to the probability of an intermediate node to be set as Modi. In other words, μ is the expected percentage of Modi nodes over all intermediate nodes in programs of the initial population. The higher the Modi Rate, the more function nodes in initial programs would be Modi.

The solution of the second point is just a uniform distribution. Cell indices of the output vector are assigned uniformly across all Modi nodes. With this solution, it is possible to

produce Modi programs that do nothing on some of the output vector cells, in which case the output value from those cells would always be zero.

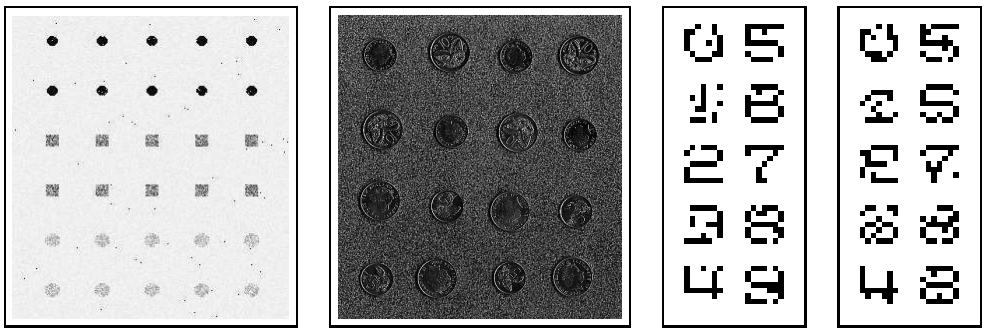
2.5 Modi Characteristics Summary

The Modi program has two major properties: 1) It can produce multiple related outputs, thus classifiers of this kind can determine the class of the input object by simply voting as in neural networks. In this way, the complex translation from a single floating point value to multiple class labels can be avoided. 2) During the evolution, the Modi program structure is just like a standard program tree, thus take the advantages of the standard GP systems.

3 Experimental Configuration

3.1 Image Data Sets

In the following experiments, we used four data sets providing object classification problems of varying difficulty. Example images are shown in Figure 4.



(a) Shapes (b) (c) (d)

Figure 4: Sample Data sets. (a) Shapes; (b) Coins; (c) Digits15; (d) Digits30.

The first set of images (Figure 4a) was generated to give well defined objects against a relatively clean background. The pixels of the objects were produced using a Gaussian generator with different means and variances for each class. Three classes of 960 small objects were cut out from those images to form the classification data set. The three classes are: black circles, grey squares, and light circles. For presentation convenience, this data set is referred to as *shape*.

The second set of images (Figure 4b) contains scanned 5 cent and 10 cent New Zealand coins. The coins were located in different places with different orientations and appeared in different sides (head and tail). In addition, the background was cluttered. We need to distinguish different coins with different sides and different rotations from the background. Five classes of 801 object cutouts were created: 160 5-cent heads, 160 5-cent tails, 160 10-cent heads, 160 10-cent tails, and the cluttered background (161 cutouts). Compared with the *shape* data set, the classification problem in this data set is much harder. Although these are still regular, man-made objects, the problem is very hard due to the noisy background and the low resolution.

The third and the fourth data sets are two digit recognition tasks, each consisting of 1000 digit examples. Each digit example is a 7×7 bitmap image object. In the two tasks, the goal is to automatically recognize which of the 10 classes (digits 0, 1, 2, ..., 9) each pattern (digit example) belongs to. Note that all the digit patterns have been corrupted by noise. In the

two tasks (Figures 4c and 4d), 15% and 30% of pixels, chosen at random, have been flipped. In data set 3, while some patterns can be clearly recognized by human eyes such as "0", "2", "5", "7", and possibly "4", it is not easy to distinguish between "6", "8"and "3", even "1" and "5". The task data set 4 is even more difficult — human eyes cannot recognize majority of the patterns, particularly "8", "9" and "3", "5" and "6", and even "1", "2" and "0". In addition, the number of classes is much larger than that in tasks 1 and 2, making the two tasks even more difficult.

For all the four data sets, the objects were equally split into three separate data sets: one third for the training set used directly for learning the genetic program classifiers, one third for the validation set for controlling overfitting, and one third for the test set for measuring the performance of the learned program classifiers.

3.2 GP System Customization

In this approach, the feature terminals consisted of four local statistical features extracted from the object cutout examples in the first two tasks, and just 49 pixel values in the third and fourth tasks. The function set consisted of the four standard arithmetic operators and a conditional operator. The division operator represents a "protected" division in which a divide by zero gives a result of zero. The conditional operator returns its second argument if its first argument is negative, and otherwise returns its third argument.

The ramped half-and-half method (Banzhaf et al., 1998; Koza, 1994) was used to generate the initial population and for the mutation operator. The proportional selection mechanism and the reproduction, crossover and mutation operators (Banzhaf et al., 1998) were used in the learning process. We used reproduction, mutation, and cross over rates of 10%, 30%, and 60%, respectively. The program depth was initialized from 3-6, and can be increased to 7 during evolution. The population size was 500. The evolutionary process was run for a maximum of 50 generations, unless it found a program that solved the problem perfectly (100% accuracy), at which point the evolution was terminated early.

3.3 Other Configurations

1. all experimental results presented are an average over 50 runs on an identical experimental setting and a different starting point.

2. The new approach (Modi-GP) will be compared with a basic GP approach (*Basic-GP*). The Basic-GP is the standard tree based GP approach, which uses Static Range Selection (Loveard and Ciesielski, 2001) as the classification strategy for converting the single output value of the learnt genetic program to class labels.

4 Results and Discussion

4.1 Overall Classification Performance

The new Modi approach and the basic GP approach are compared under the same experimental setting on the four datasets described previously. The best results are shown in Table 1. For the shape data set, both approaches did pretty well as the task is relatively easy. In particular, the Modi approach almost achieved perfect results. For the coin data set, as the task is harder, the Modi approach achieved 93.89% accuracy, 8.67% higher than the basic GP approach. For the two digit data sets, the Modi approach performed much better than the basic GP, with improvements of more than 10%. In particular, for task four, where even human eyes could only recognize a small part of the digit examples, the GP approach

with Modi program structure can achieve 54.45% on the classification accuracy, meaning that over half of the digits were correctly recognized. These results suggest that the new Modi approach can perform better than the basic GP approach for these object classification programs, particularly for relatively difficult tasks.

Table 1: A comparison of results between the new Modi approach and the basic GP approach.

Methods/ Improvement	Data Sets			
	Shape	Coin	Digit15	Digit30
Modi-GP (%)	99.77	93.89	68.11	54.46
Basic-GP (%)	99.40	85.22	56.85	44.09
Improvement (%)	0.37	8.67	11.26	10.37

Note that we also examined this approach on other data sets and the results showed a similar pattern. Details can be seen from A.

4.2 The Effect of Modi Rate μ

To investigate the effect of Modi rates in the Modi patched GP system, we did four groups of experiments on the four data sets using different Modi rates ranging from 0.0 to 1.0. Modi rate of 0.0 means that in the initial population, only the root node of each tree is a Modi node (although there could be more by the learning); Modi rate of 1.0 would mean that all non-leaf nodes are Modi nodes.

The results are shown in Figure 5 in terms of the improvement of the Modi approach over the basic GP approach. It shows that the Modi rate does affect the performance. However, the influence of Modi rates is not consistently proportional across different tasks, provided by that curves in the figure are not parallel with each other. Experiments have also shown that too big and too small Modi rates are not good, though rates in the middle range would not cause big distinction on the result of the learning. In other words, blindly pick up one modi rate in the middle range could guarantee an improvement over the basic GP approach. According to our future observation over experimenting on 15 different datasets (see appendix), a modi rate between 0.3–0.6 is a good range to start searching on. This can be considered as a heuristic for the exhaustive search over different Modi rates.

In a theoretical sense, the "middle-range modi rates are good" observation suggests that in the initial population, programs with neither too many modi nodes nor too few modi nodes are convenient for GP to *start evolving on*. This is reasonable because the interpretation of the modi rate is just the expected percentage of *reuses* and the expected times of output vector updating in the initial programs. Too low reuse and output vector updating definitely reduce the power of the program, whereas too high case makes the reusing and updating effect unnecessarily messy.

4.3 Analysis of the Results

The result of the two digit datasets are worse than the shape and the coin datasets. This is mainly due to the difficulty of the classification problem, plus that the program size (program tree depth) was set to be too small. A small program size causes two problems: (1) The evolved program would not have enough leaf nodes for handling input features, particularly when the number of input features is large, say 49 for the digits. Modi's *reusability* helps on this problem, though cannot generally fix it. 2) The probability of producing sufficient

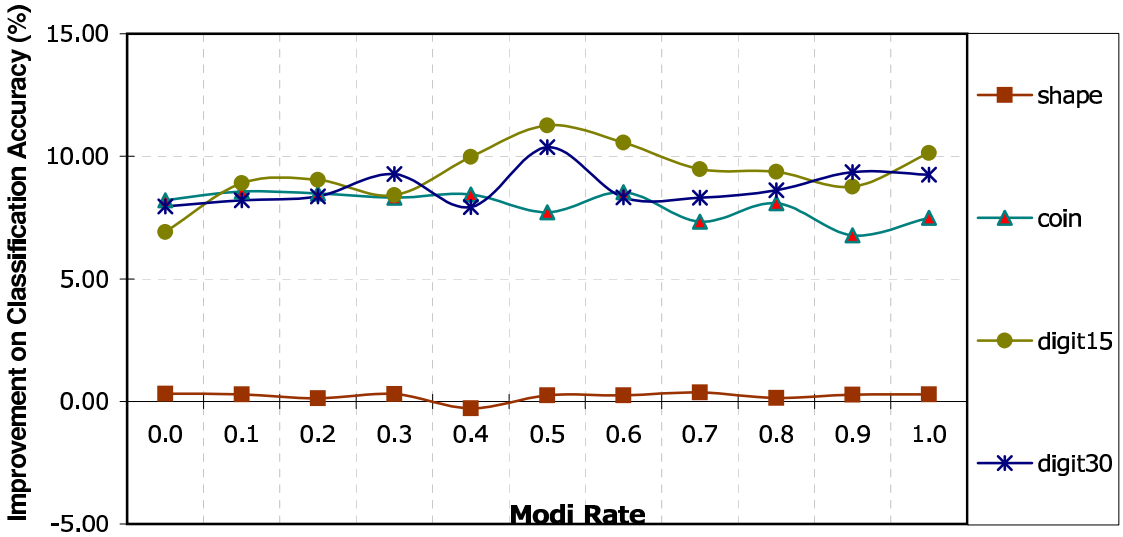


Figure 5: Effect of different Modi rates.

Modi nodes so that they all together go across the entire output vector would be very small, particularly when there are a large number of classes to be classified, say 10 for the digits. This problem reveals a disadvantage of the Modi structure, that is, it could require a larger program size than the standard program tree structure.

We originally expected a small Modi rate from 0.0 to 0.1 would lead to very bad results. However, the effect of Modi rates in the experiments was not as large as we expected. This is because we used a large size of population (500), so that the population as a whole provides sufficient Modi nodes that the *best* evolved program can use.

4.4 Efficiency Analysis

While the running times of the best evolved programs of both GP systems in testing (classification) are similar, Modi-GP is more time consuming than Basic-GP in terms of the training time. Theoretically there are two possible reasons for the slow training time: 1) the running of a single evolution is slower; 2) Modi-GP converges slower on a data set, namely it needs more evolutions to reach a satisfactory training state (say, perfect classification accuracy). Modi-GP falls into the first case. For the second one, it is actually the other way round.

Modi-GP is relatively slower for a single evolution because the most time consuming part in each evolution is to evaluate the fitness of each program in the population. However the program tree evaluating of modi tree is in general slower than the SDTree. This is because the modi nodes in the new structure have an extra task to do, which is to update the output vector.

For the second point, we happily observed that, Modi-GP actually converges much faster than Basic-GP in our experiments. For simple classification tasks such as the shape data set, Modi-GP can terminate on just a couple of evolutions. By applying the *No Free Lunch* Theorem inversely, we theoretically conclude that Modi-GP is more appropriate than Basic-GP on multiclass classification, which is consistent with our experimental results.

Although Modi-GP is able to converge faster to the optimal solution, for hard problems such as the digit data sets, in which both Modi-GP and Basic-GP are not able to terminate within the fifty generation limits, the training time of Modi-GP is higher than Basic-GP.

5 Conclusions

This paper describes a virtual program tree structure that simulates the effect of loppy directed acyclic graphs. While the programs with this structure makes multiple related outputs, the actual structure remains the standard tree and thus is naturally evolvable by the tree-based GP systems.

The structure was applied to a typical application domain that naturally fits in better with many-to-many solutions, namely multiclass classification. With the new multiple-output program tree, the awkward translation of the single number into multiple class labels was successfully avoided. This new approach was examined and compared with the basic GP approach on four object classification problems of increasing difficulty. Results showed that the new approach outperformed the basic approach on all the tasks.

The results also showed that different Modi rates lead to different results. Neither too small nor too large Modi rates were good. However, there did not seem to exist an efficient and reliable way of choosing a good Modi rate for a particular problem. Rates between 0.3 – 0.6 seemed to be a good range to start.

Although developed for object classification problems, this approach is expected to be general and can be applied to other problems, where multiple outputs are desired.

5.1 Future Work

The following points could be considered and investigated in the future:

- Note that the Modi structure gives us a *proper* subset simulation of the MRtree, but not a full set simulation. This means that the Modi structure can simulate some, but not all program structures of the MRtree. This is due to the fact that, in Modi, reuse connections (connections that cause child sharing) are simulated by the way Modi nodes pass values. Whether or not it is terribly functionally hurt needs to be further investigated in the future.
- For digit tasks, we will investigate a larger program size and examine whether the performance can be improved.
- We will investigate other solutions of the Modi node distribution problem other than just a uniform distribution. It is worth considering whether it is better to add more heuristic controls in the solution for fairer assignment.
- We would also like to investigate ways of extending the Modi idea to evolve neural networks and belief networks, thus producing a kind of *Skinnerian creature* other than the pure GP based *Darwinian creature*.

Acknowledgements

We would like to thank Malcolm Lett and Will Smart in our genetic programming group and members in the AI research group particularly Peter Andreae and Marcus Frean for a number of useful discussions.

References

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Ap-*

plications. San Francisco, Calif. : Morgan Kaufmann Publishers; Heidelburg : Dpunkt-verlag, 1998. Subject: Genetic programming (Computer science); ISBN: 1-55860-510-X.

Daniel Howard, Simon C. Roberts, and Richard Brankin. Target detection in SAR imagery by genetic programming. *Advances in Engineering Software*, 30:303–311, 1999.

John R. Koza. *Genetic Programming I: on the programming of computers by means of natural selection*. Cambridge, Mass. : MIT Press, London, England, 1992.

John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, Mass. : MIT Press, London, England, 1994.

Nicholas J. Hopper and Mitchell L. Reierson. Genetic Programming: Impact of types on essentially typeless problems in GP University of Minnesota, Morris.

Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

D.J.C. MacKay: *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press (2003).

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP research group, editors, *Parallel distributed Processing, Explorations in the Microstructure of Cognition, Volume 1: Foundations*, chapter 8. The MIT Press, Cambridge, Massachusetts, London, England, 1986.

Will Smart and Mengjie Zhang. Classification strategies for image classification in genetic programming. In Donald Bailey, editor, *Proceeding of Image and Vision Computing Conference*, pages 402–407, Palmerston North, New Zealand, November 2003.

Mengjie Zhang, Victor Ciesielski, and Peter Andreae. A domain independent window-approach to multiclass object detection using genetic programming. *EURASIP Journal on Signal Processing, Special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis*, 2003(8):841–859, 2003.

Mengjie Zhang and Will Smart. Multiclass object classification using genetic programming. In Guenther R. Raidl, Stefano Cagnoni, Jurgen Branke, David W. Corne, Rolf Drechsler, Yaochu Jin, Colin Johnson, Penousal Machado, Elena Marchiori, Franz Rothlauf, George D. Smith, and Giovanni Squillero, editors, *Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC*, volume 3005 of *LNCS*, pages 367–376, Coimbra, Portugal, 5-7 April 2004. Springer Verlag.

A Further Experiments and Results

This appendix describes more results of our new Modi approach compared with the standard GP approach on additional classification data sets. These results shows exactly the same patterns, which strongly support the conclusions previously made by additional evidence.

A.1 Additional Data Sets

For further testing the behaviour of Modi patched GP, we examined and compared the new Modi GP approach with the basic GP approach over 15 data sets. These include two simple shape data sets (`squ-3` and `squ-4`), four New Zealand coin data sets (`c-5c, c-10c, cEasy` and `cHard`), and nine digit data sets (`dig00, dig05, dig10, dig15, dig20, dig30, dig40, dig50` and `dig60`).

The six object classification tasks in the two shape and the four coin data sets are similar to those presented in section 3. Each of the nine digit recognition tasks involves a file (a collection) of binary digit images. Each file contains 100 examples for each of the 10 digits (0, 1, ..., 9), making a total number of 1000 digit examples. Each digit example is an image of 7×7 bitmap. These tasks were chosen to provide classification problems of increasing difficulty. In all of these recognition problems, the goal is to automatically recognize which of the 10 classes (digits 0, 1, 2, ..., 9) each pattern (digit example) belongs to. Except for the first file which contains clean patterns, all data patterns in the other eight files have been corrupted by noise. The amount of noise in different files was randomly generated based on the percentage of flipped pixels and was given by the two numbers `nn` in the data set names, as described above.

Figure 6: Sample images in the digit recognition tasks.

Sample images/objects in the nine digit data sets are shown in Figure 6. The nine lines of digit examples correspond to the recognition tasks in the nine data sets. The first three tasks, one with clean data and two with only 5% and 10% of flipped rates, are relatively straightforward for human eyes, though there is still some difficulty in distinguishing between

"3" and "9". With the increase of the flipped rate in these patterns such as task 4 and task 5, it becomes more difficult to classify these digit patterns, even if humans can still recognize the majority. From task 6 to task 9, however, it is very difficult, even impossible, for human eyes to make good discrimination.

A.2 Results on All 15 Data Sets

The results on the 15 data sets are shown as plotted in Figure 7, as an overall view of the effectiveness of Modi-GP compared with Basic-GP in terms of the classification accuracy. The bottom square labelled curve is for Basic-GP; The top clusters are Modi-GP with different modi rates μ, as labelled. It can be seen from these curves, Modi-GP generally performs better than Basic-GP, presented as the cluster is roughly above the Basic-GP curve. Modi rate does affect the performance as the curves with different Modi rates are clustered but not strictly overlapped. However, the influence is not much comparing with the improvement from Basic-GP, shown by the big gap between the clustered curve and the Basic-GP curve.

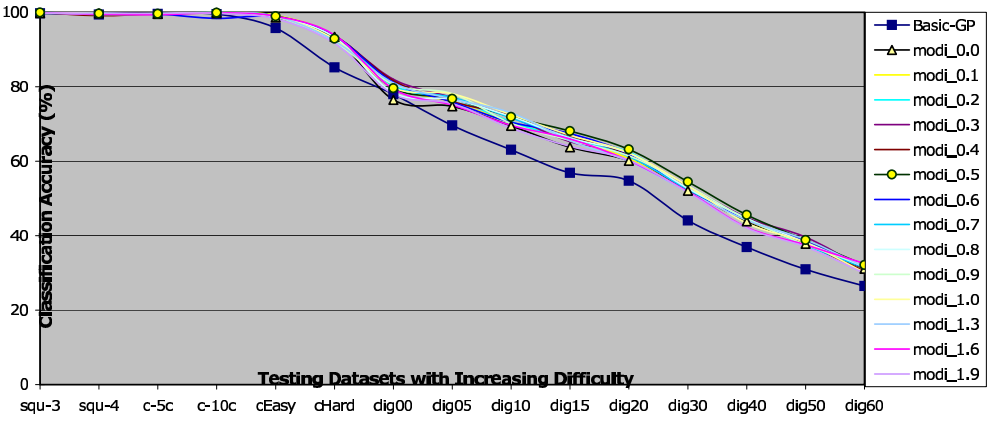
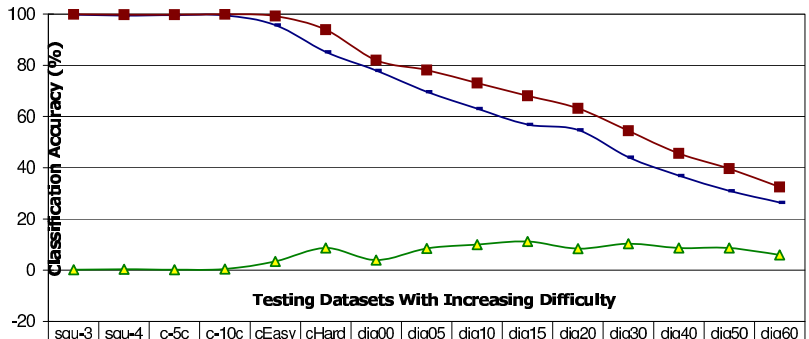


Figure 7: Effectiveness: overall view

Figure 8 shows a comparison of the best results achieved by the two GP approaches. The improvement of the Modi GP with the best Modi rates over Basic-GP is also shown by the bottom curve, called the *difference curve*. The precise data value is also presented, shown in the table beneath the graphical region.



	squ-3	squ-4	c-5c	c-10c	cEasy	cHard	dig00	dig05	dig10	dig15	dig20	dig30	dig40	dig50	dig60
BasicGP	99.71	99.40	99.57	99.47	95.74	85.22	78.00	69.63	63.07	56.85	54.76	44.09	36.92	30.95	26.47
ModiGP: best mu	99.94	99.77	99.74	99.91	99.17	93.89	82.00	78.14	73.04	68.11	63.16	54.46	45.58	39.61	32.49
Diff: ModiGP - BasicGP	0.23	0.37	0.17	0.44	3.43	8.67	4.00	8.51	9.97	11.26	8.40	10.37	8.66	8.66	6.02

Figure 8: Modi-GP with best and worst modi rates for each classification task.

The results are very consistent with those presented in the main text, which support our conclusions.