Online Program Simplification in Genetic Programming

Mengjie Zhang[1,2], Phillip Wong[1], and Dongping Qian[2]

[1] School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand
[2] Artificial Intelligence Research Centre, Agricultural University of Hebei, China
{mengjie, phillip}@mcs.vuw.ac.nz, {zmj, qdp}@hebau.edu.cn

Abstract. This paper describes an approach to online simplification of evolved programs in genetic programming (GP). Rather than manually simplifying genetic programs after evolution for interpretation purposes only, this approach automatically simplifies programs during evolution. In this approach, algebraic simplification rules, algebraic equivalence and prime techniques are used to simplify genetic programs. The simplification based GP system is examined and compared to a standard GP system on a regression problem and a classification problem. The results suggest that, at certain frequencies or proportions, this system can not only achieve superior performance to the standard system on these problems, but also significantly reduce the sizes of evolved programs.

1 Introduction

Since the late 1990s, genetic programming (GP) has already been applied to many fields, including image analysis [1], object detection [2], regression problems [3] and even control programs for walking robots [4], and achieved quite a reasonable level of success.

While showing promise, current GP techniques are limited, often require a very long evolution time, and frequently do not give satisfactory results for difficult tasks. One problem is the redundancy of programs. Typically, the programs are not simplified until the end of the evolutionary process to enable analysis. However, the redundancies also affect the search process. They force the search into exploring unnecessarily complex parts of the search space [5,2]. The redundancies and complexities have the undesirable consequences that the search process is very inefficient in execution, and the programs are very difficult to understand and interpret. However, the redundant components of the evolving programs can provide a wider variety of possible program fragments for constructing new programs.

The goal of this paper is to invent a method in GP that does online program simplification during the evolutionary process. We will investigate the effect of performing online simplification of the programs during the evolutionary process, to discover whether the reduction in complexity outweighs the possible benefits of redundancy. This approach will be examined and compared with the standard GP without simplification on a regression problem and a classification problem.

Table 1. Typical simplification rules

No.	Precondition	Effective Result	No.	Precondition	Effective Result
(1)	if<0(A, b, c)	→ b if A < 0, else c	(2)	if<0(a, b, b)	→ b
(3)	A + B	→ C, C = A + B	(4)	A - B	→ C, C = A - B
(5)	A × A	→ C, C = A × B	(6)	A ÷ B	→ C, C = A ÷ B
(7)	A + (B + c)	→ C + c, C = A + B	(8)	A + (B - c)	→ C - c, C = A + B
(9)	A - (B + c)	→ C - c, C = A - B	(10)	A - (B - c)	→ C + c, C = A - B
(11)	A × (B × c)	→ C × c, C = A × B	(12)	A × (B ÷ c)	→ C ÷ c, C = A × B
(13)	A ÷ (B ÷ c)	→ C × c, C = A ÷ B	(14)	A + (b + C)	→ B + b, B = A + C
(15)	A + (b - C)	→ B + b, B = A - C	(16)	A - (b + C)	→ B - b, B = A - C
(17)	A - (b - C)	→ B - b, B = A + C	(18)	A × (b × C)	→ B × b, B = A × C
(19)	A × (b ÷ C)	→ C × b, B = A ÷ C	(20)	A ÷ (b ÷ C)	→ B ÷ b, B = A × C
(21)	a ÷ 1	→ a	(22)	a ÷ a	→ 1
(23)	0 ÷ a	→ 0	(24)	0 × a = a × 0	→ 0
(25)	a × 1 = 1 × a	→ a	(26)	a + 0 = 0 + a	→ a
(27)	a - 0	→ a	(28)	a - a	→ 0
(29)	$a \times \frac{1}{b} = \frac{1}{b} \times a$	→ $\frac{a}{b}$	(30)	$a \times \frac{b}{a} = \frac{b}{a} \times a$	→ b

2 The Approach

In the tree-based GP, a genetic program looks like an algebraic expression. The function set consists of the commonly used four arithmetic operators and a conditional operator `+, -, ×, ÷, if`. The terminal set consists of a number of feature terminals from the task and several constant terminals. The task of the simplification method is to obtain a smaller program, by removing the redundancy of a program, that yields the same output as the original program. In this approach, we use this idea to construct simplification rules, apply these rules using a postfix search to the genetic programs, and use hashing to estimate the algebraic equivalence to simplify the genetic programs during evolution.

2.1 The Simplification Rules

As in algebraic expression simplification, we use multiple rules (*ruleset*) to simplify a given genetic program, as shown in table 1. A specific rule might only be suitable for removing/reducing a particular part of the genetic program. In this table, constants are represented by upper-case letters (e.g. `A`, `B`), and variables are represented by lower-case letters (e.g `a`, `b`).

2.2 Algebraic Equivalence of Two Subtrees

In a simplification system, it is important to determine whether two subtrees have the same role (or are *equivalent*). For two single nodes, this is fairly trivial; for multi-node subtrees, this will be more difficult. Our goal is to allow for not only noticeably similar expressions (e.g. `(x + y + z)` and `(z + x + y)`) to be identified as equivalent, but also seemingly dissimilar expressions, for example, `(/ (+ (- (* w x) (* x y)) (* (- w y) y)) (- (* x x) (* y y)))` and `(/ (- w y) (- x y))` as well, which is a hard problem.

We use hashing techniques to address the equivalence of two subtrees [6,7] to cope with all common terminals and functions in the evolved programs. In this work, p is used to denote the *hashing order* for the hash function. It is important that the collection of hash values qualify as a finite field [8] and so p should be a prime number as any finite field with p elements is isomorphic to $\mathbb{Z}_p$ [8].

Feature Terminals. In GP, feature terminals represent inputs from the task environment and always keep the same value for a particular fitness case for all genetic programs during evolution. Accordingly, in this approach, we assign the feature terminals certain random hash values at the beginning of a GP run, which remain unchanged for the entire evolution.

Constant Terminals. In GP, constants are usually represented by floating point numbers. We handle this by approximating the floating point with a rational number, thus converting it to a simple division of two integers.

Calculating accurate and irreducible rationals can be very time consuming, so a quick approximation is used. The numerator is formed by multiplying the floating point by a predefined precision constant (δ) and truncating the leftover fractional part. Using the same precision constant as a denominator, a rational representation can be very quickly found.

$$Hash(c) = \frac{c \times \delta}{\delta} \bmod p = (c \times \delta) \times \frac{1}{\delta} \bmod p \tag{1}$$

This approach, of course, requires *modular division.* Now, the division of two numbers $\frac{x}{y}$ is equivalent to the multiplication of the first number with the multiplicative inverse of the second number $x \times \frac{1}{y}$. So to perform division, one needs only to calculate the multiplicative inverse of y and multiply by x. The key point here is to find the integer equivalence of the inverse of $\delta \bmod p$. In this approach, this is done using the *Extended Euclidean Algorithm* [9,10].

The Arithmetic Operators. Because the hashing method takes place in a finite field, hashing these arithmetic operators are handled using *modulo arithmetic* within the field as follows, where the division hashing follows the rule of the extended Euclidean algorithm mentioned earlier.

$$Hash(A + B) = (A + B) \bmod p \tag{2}$$

$$Hash(A - B) = (A - B) \bmod p \tag{3}$$

$$Hash(A \times B) = (A \times B) \bmod p \tag{4}$$

$$Hash(A \div B) = (A \div B) \bmod p \tag{5}$$

The `if` operator. The `if` conditional operator is a more difficult case, as it is *not* an arithmetic function and so cannot simply be converted to a *modulo arithmetic* equivalent. We use the following approach to handle this operator, which uses division and addition to take into account the position of the three parameters.

$$Hash(\texttt{if}(A, B, C)) = (\frac{A}{B} + C) \bmod p \tag{6}$$

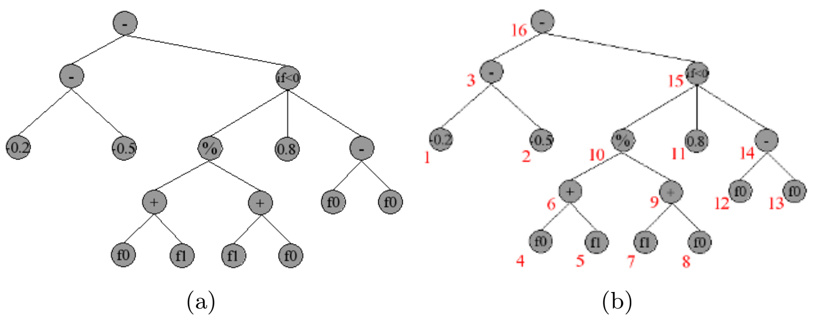

Fig. 1. An example program. (a) The original program tree; (b)traversal order.

Operator Closure. All of the functions supported are closed, meaning that for any of the functions $\diamond \in \{+, -, \times, \div, \texttt{if} < 0\}$, $(Hash(A) \diamond Hash(B)) \bmod p = Hash(A \diamond B)$ in $\mathbb{Z}_p$. Using this property, one does not need to recalculate the hash values of subtrees each time a tree is to be hashed, as stored hash values of subtrees can be combined to give correct hash values of the whole tree.

2.3 Simplification Process

To apply the ruleset to a genetic program for simplification, we use a kind of "greedy" engine, which is a recursive algorithm. It recursively travels through the program tree in a bottom-up fashion by the postfix order traversal mode. For each node it processes, the algorithm checks each simplification rule in the ruleset. If a rule matches, it is applied to the partial tree associated with the node to make simplification. If *none* of the rules can be applied at a node, the algorithm moves to the next (either neighbouring or parent) node. In this way, the algorithm guarantees that each node in the program tree is visited once only.

Here, we use an example to show the simplification process for a given genetic program. The example program `(- (- -0.2 -0.5) (if<0 (% (+ f0 f1) (+ f1 f0)) 0.8 (- f0 f0)))` can be represented in the tree shown in figure 1 (a). Assume that the hashing order is `17`, `f0` and `f1` are "randomly" assigned the values `3` and `5` respectively. The algorithm traverses the program tree in a "bottom-up" fashion using a post-fix traversal. This means that the algorithm processes the program nodes in the order depicted by integers in figure 1 (b).

The first node inspected by the algorithm is "`-0.2`", followed by "`-0.5`". As no simplification rule exists in the ruleset that governs single nodes, these nodes (and indeed the entire bottom layer of nodes) are left unchanged. Next, the algorithm moves to the parent node of "`-0.2`" and "`-0.5`", which is "`-`". The subtree formed by this node and its children `(- -0.2 -0.5)` matches the precondition for rule (4) `A—B`. The system applies this rule, replacing the subtree with the rule's effective result: "`0.3`".

Now, the subtrees `(+ f0 f1)` and `(+ f1 f0)` do not match the preconditions for any of the rules, so are left unchanged. Note however, that they both have the same algebraic equivalence hash value (shown in figure 2 (a)). Therefore, when node *10* ("%") is inspected, the subtree `(% (+ f0 f1) (+ f1 f0))` does indeed match the precondition for rule (22) `a÷a`. The entire subtree is replaced using the rule to a single node `1`. Similarly, the subtree `(- f0 f0)` matches rule (28) `a−a` and is replaced by the single node `0` when the algorithm processes "`-`". Figure 2 (b) shows the tree after processing nodes *1* through *14*.

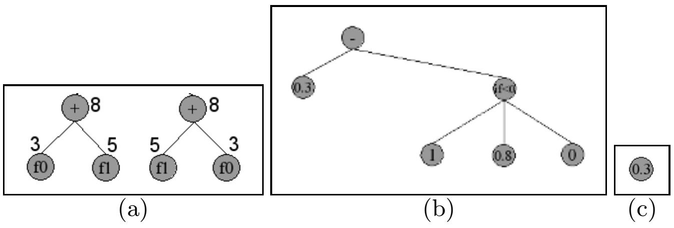

Fig. 2. Program simplification. (a)Hashing of two subtrees with same value; (b) Partial simplification; (c) final program.

At this stage, the program is already reduced to 6 nodes in size, and there are still two nodes left to be processed. Inspecting the `if<0` node, the algorithm matches it with rule (1) `if<0(A b c)`, as the first parameter of the `if<0` operator is a constant. In this case, the constant is 1, which will obviously never be less than 0. The system then, following the rule, replaces this subtree with its third parameter, which is 0.

Lastly the root node is processed, which again matches rule (4) `A−A`. Applying it yields the final result, a single numerical constant node "`0.3`" (figure 2 (c)).

3 Experimentation Setup

3.1 Data Sets

We used two data sets, a symbolic regression task and a object classification tasks, to examine the simplification method. The regression task is a quite complicated piecewise function. We used 200 data points in [-10, 10] as the fitness cases and the task is to evolve a genetic program that conform the curve, as shown in figure 3 (a).

The classification task uses a subset of the Yale Database B Face Dataset [11]. It consists of face images of 5 subjects taken from a single position under 65 different lighting conditions. This creates a set of 325 instances. The backgrounds of the faces are very complicated and different, making the classification problem more difficult. Example images for the task are shown in figure 3 (b). Due to a small number of data examples, 10-fold cross validation method is applied.

3.2 Terminal Set, Function Set and Fitness Function

The terminal set consists of a number of *feature terminals* as well as several randomly generated *constant terminals*. For the symbolic regression task, the feature

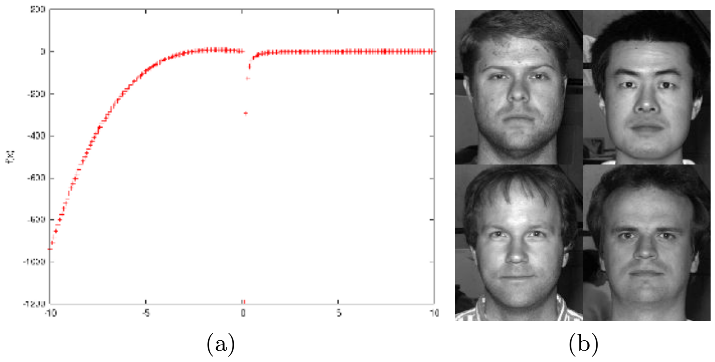

Fig. 3. Two data sets. (a) regression; (b) classification.

terminal corresponds to the single independent variable. In the face data set, we used 18 feature terminals representing the extracted pixel statistic features from the various facial regions. We use the four basic arithmetic operators and a conditional operator to form the function set $\{+, -, \times, \div, \texttt{if<0}\}$. For the symbolic regression task, the fitness of a program is governed by the mean squared error of the desired output and the actual output of the program on all the fitness cases. For the classification task, the fitness of a program is governed by the classification accuracy in the training set.

3.3 Parameters

The population size is 500, the rates used for crossover, mutation and reproduction are 60%, 30% and 10%, respectively, and the maximum program depth is 8. The evolution will run 50 generations unless an ideal solution program is found, in which case the evolution was terminated early. The *hash order* p was 1000077157, the *constant precision* δ is 1000000. In addition, we used different *proportions*, the percentage of programs in a population to be applied to simplification, and different *frequencies*, how often (in generations) the simplification process is applied, to examine property of the simplification algorithm. The proportions tested here are 0%, 5%, 10%, 20%, 50% and 100%. The frequency used here are every 0, 1, 2, 4, and 6 generations. All single experiments were repeated 50 runs to get the means and standard deviations as results.

4 Results and Discussion

4.1 Overall Results

Table 2 shows the typical best results of the two GP approaches on the two data sets in terms of the effectiveness (best fitness—mean squared error for regression and classification accuracy for classification), training efficiency (training time), and average size of all the programs in the systems in number of nodes.

Table 2. Average best results for the two tasks

Task	Frequency	Proportion	Best Fitness	Time(s)	Avg. Prog Size
Regression	Without	Without	83.774 ± 75.283	5.141 ± 1.019	104.436 ± 22.171
	Every 1	20%	60.354 ± 44.365	4.743 ± 1.383	77.392 ± 24.393
	Every 2	100%	67.346 ± 59.315	4.270 ± 0.759	74.841 ± 13.886
Classification	Without	Without	$85.5\% \pm 11.7\%$	2.646 ± 0.578	37.861 ± 8.755
	Every 1	20%	$87.3\% \pm 7.2\%$	2.445 ± 0.484	29.436 ± 5.855
	Every 2	100%	$86.7\% \pm 11.7\%$	2.367 ± 0.460	29.364 ± 5.966

Efficiency. As expected, it is always possible to find certain proportions and/or frequencies at which the GP approach with the simplification spent shorter time to evolve good programs than the standard GP without simplification. This is mainly because the simplification process removes the redundancy, makes the genetic programs shorter, and accordingly reduces the search space.

Effectiveness. According to table 2, it is always possible to find certain proportions or frequencies at which the GP approach with the proposed simplification achieved superior fitness, either in mean square error or accuracy, on these data sets than the basic GP approach without simplification.

We hypothesised that the simplification process during evolution might destroy the existing good building blocks of the genetic programs, which might result in worse performance. However, these results are clearly different from the original hypothesis. After checking the evolutionary process, we identify the following reasons. At the beginning of evolution, although the simplification algorithm might destroy some potentially good building blocks, this effect was very much offset by the powerful crossover operator, which can preserve good, even form larger, building blocks. At the later stage, when the programs are getting larger, the crossover operator starts to destroy good existing building blocks. The simplification algorithm, however, can generate new genetic materials which might contain good building blocks by *reorganising* the entire genetic programs.

Program Size. According to our experiments, the average size of the programs is significantly reduced for the GP system with simplification at all frequencies and all proportions over the basic GP without simplification. The small size programs have a big advantage in that the actual computation time of the solution program will be short. This is particularly useful in the situations that has a strict time requirement such as in some industrial control and security systems.

4.2 Simplification Frequency/Proportion Analysis

According to our experiments, applying simplification to *all* programs at *every* generation (the last line of table 2) led to a slight loss in fitness and/or a slightly higher computational cost in most cases. This suggests that, if we apply the simplification too often, the programs will not have sufficient chances for evolution and the simplification overhead will be increased. If the evolution chances are reduced to some extent or the overhead outweighs the time saved from processing smaller simplified programs, the performances will deteriorate.

5 Conclusions

This paper aimed to develop an online program simplification approach in GP during the evolutionary process. This goal was successfully achieved by defining a set of algebraic simplification rules, traversing the program tree in a bottom-up fashion by a postfix order, and applying the simplification rules along with an algebraic equivalence component to non-terminal nodes in the evolved programs.

The GP system with the simplification algorithm was examined and compared with the basic GP approach without simplification on a regression problem and a classification problem. The results suggest that, at certain proportions or certain frequencies, the new simplification approach always outperformed the basic GP approach in terms of system effectiveness, efficiency and program size on these data sets.

The results also suggest that performing simplification of all programs in the population at every generation is not recommended. While the approach seems to be able to reduce the search space, it is not clear whether and/or how it destroys good building blocks in the early stage of evolution, which needs to be further investigated.

Acknowledgement. This work was supported in part by the Marsden Fund at Royal Society of New Zealand under grant No. 05-VUW-017 and University Research Fund 6/9 at Victoria University of Wellington.

References

1. Poli, R.: Genetic programming for image analysis. In Koza, J.R., Goldberg, D.E., Fogel, D.B., RioloOB, R.L., eds.: Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University, CA, USA, MIT Press (1996) 363–368
2. Zhang, M., Ciesielski, V.: Genetic programming for multiple class object detection. In Foo, N., ed.: 12th Australian Joint Conference on Artificial Intelligence. Volume 1747 of LNAI., Sydney, Australia, Springer-Verlag (1999) 180–192
3. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
4. Busch, J., Ziegler, J., Aue, C., Ross, A., Sawitzki, D., Banzhaf, W.: Automatic generation of control programs for walking robots using genetic programming. In: EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming, London, UK (2002) 258–267
5. Tackett, W.A.: Genetic programming for feature discovery and image discrimination. In Forrest, S., ed.: Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, University of Illinois at Urbana-Champaign, Morgan Kaufmann (1993) 303–309

6. Martin, W.A.: Determining the equivalence of algebraic expressions by hash coding. j-J-ACM **18**(4) (1971) 549–558
7. Gonnet, G.H.: Determining equivalence of expressions in random polynomial time. In: STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (1984) 334–341
8. Lidl, R., Niederreiter, H.: Introduction to finite fields and their applications. Cambridge University Press, New York, NY, USA (1986)
9. Trappe, W., Washington, L.C.: Introduction to Cryptograpy with Coding theory. 2ed edn. Prentice-Hall (2006)
10. Cherowitzo, B.: (2006) Lecture Notes. http://www-math.cudenver.edu/~ wcherowi/courses/m5410/exeucalg.html. Visited on 7 January 2006.
11. Georghiades, A., Belhumeur, P., Kriegman, D.: From few to many: Illumination cone models for face recognition under variable lighting and pose. IEEE Trans. Pattern Anal. Mach. Intelligence **23**(6) (2001) 643–660