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

School of Mathematics, Statistics and Computer Science
Computer Science

Algebraic Simplification of Genetic Programs during Evolution

Phillip Wong, Mengjie Zhang

Technical Report CS-TR-06/7
February 2006

School of Mathematics, Statistics and Computer Science
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 Mathematics, Statistics and Computer Science
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

Algebraic Simplification of Genetic Programs during Evolution

Phillip Wong, Mengjie Zhang

Technical Report CS-TR-06/7
February 2006

Abstract

Program bloat is a fundamental problem in the field of Genetic Programming (GP). Exponential growth of redundant and functionally useless sections of programs can quickly overcome a GP system, exhausting system resources and causing premature termination of the system before an acceptable solution can be found. Simplification is an attempt to remove such redundancies from programs. This paper looks at the effects of applying an algebraic simplification algorithm to programs during the GP evolution. The GP system with the simplification is examined and compared to a standard GP system on four regression and classification problems of varying difficulty. The results suggest that the GP system employing a simplification component can achieve superior efficiency and effectiveness to the standard system on these problems.

Keywords Genetic Programming, Algebraic Simplification, Program Simplification, Code Bloating, Online Simplification.

Author Information

Both authors are academic staff members and postgraduate students in computer science in the School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, New Zealand.

1 Introduction

Genetic programming (GP) [8] is a method of automatically generating programs for solving specific tasks. Firstly, an initial group of randomly generated *genetic* programs, normally represented as parse trees such as LISP-S trees, is created. The process of selection based on fitness is carried out to provide a basis for the next program generation. Fitness is determined by running the programs and evaluating them on a set of criteria called *fitness function*. The genetic operators of *crossover* for swapping of sections of programs, *mutation* for random alterations to a program, and *reproduction* for retaining the best programs are applied to the selected programs to create a new population of programs. The process of creating new generations is repeated until certain termination criterion is met. The *"best"* program in the last generation is usually used as the resulting system solution. GP can be seen as a genetic beam search through the space of possible solutions to the task.

GP is an emerging field in evolutionary computing and machine learning and has already been applied to many tasks, including image analysis [13], object detection [16], regression problems [8] and even control programs for walking robots [3]. GP has been very successful in solving or performing these tasks and "now routinely delivers high-return human-competitive machine intelligence" [9].

However, one of the current and fundamental problems in GP is that the process of genetic programming will inevitably introduce some redundancy into the evolved programs [8, 14, 2]. This redundancy is regarded as a fundamental problem of GP as it slows down the search process by consuming large amounts of memory and causes exploration of large unnecessary parts of the search space. The search process continues to slow as the programs become larger until the programs become too large for the system's memory to hold, halting the system before a "good" solution can be found. Redundancy can also result in an unnecessarily complex program, which is inefficient in its execution and difficult to interpret and comprehend.

On the other hand, this redundancy may aid the effectiveness of the evolutionary process by providing a more diverse selection of program fragments for the process to use, and protecting useful "building blocks" within programs from the destructive nature of the crossover operator [2].

Simplification is a process applied in order to reduce the complexity of an expression, as well as eliminate any superfluous details. Simplification can be implemented in various ways, including using simple algebraic techniques, translation into canonical forms or numeric hashing techniques. Typically simplification is applied at the end of the evolutionary process to remove some of the complexity of the program, reducing the resource usage and improving comprehensibility, enabling it to run faster and to be easier to interpret. The editing operation proposed in [8] is an example of this kind. But as program redundancy is a problem which also occurs *during* the evolutionary process, online simplification during evolution to improve performance in the whole system needs to be investigated.

1.1 Goals

The goal of this paper is to invent a method in GP that does 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 four regression and classification problems of increasing difficulty. Specifically, we are interested in:

- how the online simplification algorithm can be constructed by combining some algebraic simplification rules and hashing techniques;

- whether the simplification improves the system efficiency of the evolutionary process; and
- whether this approach deteriorates the classification performance compared with the standard GP method without simplification.

1.2 Structure

The rest of the paper is organised as follows. Section 2 describes the simplification algorithm developed in this paper. Section 3 presents the four data sets and experiment configurations. Section 4 describes the results with discussions. Section 5 concludes and gives future work directions.

2 The Program Simplification Approach

In the standard GP system, the programs are represented as a LISP-S (or similar language) expression, which is stored in a tree representation [8]. The ramped half-and-half method was used for generating programs in the initial population and for the mutation operator [1]. The proportional selection mechanism and the reproduction, crossover and mutation operators [8] were used in the learning and evolutionary process. The function set consists of the commonly used four arithmetic operators $(+, -, \times, \div)$ and an *if* (conditional) operator. The terminal set consists of a number of feature/variable terminals from the task and several constant terminals. Based on this setting, a genetic program looks like an algebraic expression.

The new approach introduced in this paper uses the same setting as the standard GP approach mentioned above. The major difference between them is that the new approach has an online program simplification algorithm to be applied to the genetic programs during the evolutionary process.

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 the idea in the algebraic expression simplification 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.

In the rest of the section, we describe the simplification rules and the simplification process, then give an example to show how a program can be simplified, and finally summarise the simplification algorithm.

2.1 The Simplification Rules

As in algebraic expression simplification, we use multiple rules to simplify a given genetic program. A specific rule might only be suitable for removing/reducing a particular part of the genetic program.

Similarly to STRIPS operators [5], we use two parts, a *precondition* and a *postcondition* to represent the simplification rules. The precondition represents the state of the surrounding nodes in the program tree that must be present in order to be able to apply the simplification rule, and the postcondition represents the additions and deletions made to the program tree to obtain the simplified form.

These rules form the *ruleset* of the program simplification algorithm, which covers major sources of redundancy in the evolved genetic programs. For example,

- an arithmetic operator with only constant children (e.g. `(+ 3 2) = 5`)

- subtraction or division of self (e.g. `(- f0 f0) = 0`)
- redundant conditionals, where the outcome is always the same (e.g. `if<0(2 f0 f1) = f1`)

A selection of the simplification rules used in this approach are presented in table 1. In this table, constants are represented by lower-case characters (e.g. `a`, `b`, `x`, `j`), and variables are represented by upper-case characters (e.g `A`, `B`, `X`, `J`).

Table 1: The simplification rules used.

Precondition		Effective Result
`if<0(a, B, C)`	$\rightarrow$	`B if a` $<$ `0, else C`
`if<0(A, B, B)`	$\rightarrow$	`B`
`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 + (b + C)`	$\rightarrow$	`c + C, c = a + b`
`a + (b - C)`	$\rightarrow$	`c - C, c = a + b`
`a - (b + C)`	$\rightarrow$	`c - C, c = a - b`
`a - (b - C)`	$\rightarrow$	`c + C, c = a - b`
`a` $\times$ `(b` $\times$ `C)`	$\rightarrow$	`c` $\times$ `C, c = a` $\times$ `b`
`a` $\times$ `(b` $\div$ `C)`	$\rightarrow$	`c` $\div$ `C, c = a` $\times$ `b`
`a` $\div$ `(b` $\div$ `C)`	$\rightarrow$	`c` $\times$ `C, c = a` $\div$ `b`
`a + (B + c)`	$\rightarrow$	`b + B, b = a + c`
`a + (B - c)`	$\rightarrow$	`b + B, b = a - c`
`a - (B + c)`	$\rightarrow$	`b - B, b = a - c`
`a - (B - c)`	$\rightarrow$	`b - B, b = a + c`
`a` $\times$ `(B` $\times$ `c)`	$\rightarrow$	`b` $\times$ `B, b = a` $\times$ `c`
`a` $\times$ `(B` $\div$ `c)`	$\rightarrow$	`b` $\times$ `B, b = a` $\div$ `c`
`a` $\div$ `(B` $\div$ `c)`	$\rightarrow$	`b` $\div$ `B, b = a` $\times$ `c`
`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`
`A` $\times$ $\frac{1}{B}$ `=` $\frac{1}{B}$ $\times$ `A`	$\rightarrow$	$\frac{A}{B}$
`A` $\times$ $\frac{B}{A}$ `=` $\frac{B}{A}$ $\times$ `A`	$\rightarrow$	`B`

2.2 The Simplification Process

To apply the ruleset to a genetic program for simplification, we used 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 the precondition for 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 node (either neighbouring node or parent node).

In this way, the algorithm guarantees that each node in the program tree is only visited once. However, as all simplification rules only look at a static and limited area, arbitrary depth levels of simplification (simplification of terms which are not neighbouring but far away) are not supported in this algorithm.

2.3 Estimating Algebraic Equivalence

Another important aspect of a simplification system is determining when `X` is equal to `Y`, providing a mechanism for evaluating rules in the ruleset. This is fairly trivial when comparing single nodes, as one needs simply to check whether the nodes are identical. However, checking whether multi-node subexpressions or subtrees are equal is 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.

In this approach, we use hashing to address the algebraic equivalence of two subtrees/subprograms. The hashing function is used to extend the algebraic system mentioned earlier to be capable of simplifying more expressions. In the 1970s and 1980s, [12] and [7] describe methods for achieving algebraic equivalence using hashing methods for algebraic expressions. In this approach, we use a variant of those methods to cope with all common terminals and functions in the evolved genetic programs.

Note that using the hashing technique to determine algebraic equivalence adds to the risk of two *non-equivalent* subexpressions being determined as equal and one or both being discarded (depending on the simplification rule). By using a very large number of distinct hash values, the number of *collisions* can be kept minimal, and probabilistically minute. In this work, p is used to denote the *hashing order* for the hash function (i.e. the total number of possible hash values). It is important that the collection of hash values qualify as a finite field ([10]) and so p should be a prime number. Note that any finite field with p elements is isomorphic to $\mathbb{Z}_p$ [10] (integers from 0 to p).

In the rest of this subsection, we describe how to estimate the algebraic equivalence for feature terminals, constant terminals, the four arithmetic operators and the conditional operator.

2.3.1 Feature Terminals

In a GP system, a feature terminal represent inputs from the task environment, such as an image feature in object classification or a simple variable in symbolic regression. The *important* attribute of these terminals is that a feature terminal always keeps the same value for a particular fitness case for all genetic programs during the evolutionary process. Accordingly, in this approach, the feature terminals are assigned random hash values at the beginning of the GP system run and remain unchanged for the entire duration of the evolutionary process. Specifically, we use:

$$Hash(Feature_n) = \text{a random value in } \mathbb{Z}_p \tag{1}$$

2.3.2 Constant Terminals

In a GP system, constants can be any numeric type: integers, rationals, floating point, etc. Therefore, the hash function needs to be designed to handle all these types, of which the

most difficult is floating point. [12] does not describe a solution to this in his paper. In this approach, 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.

$$Hash(c) = \frac{c \times \delta}{\delta} \bmod p = (c \times \delta) \times \frac{1}{\delta} \bmod p \tag{2}$$

This approach of course, requires modular division which one may not be familiar with. 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* [15, 4]. For any two integers a and b, there exists two integers q, r such that $a = b \cdot q + r$. Commonly, q is called the quotient and r the remainder. Starting from step 0, the algorithm additionally calculates an auxiliary number x_i at each division step, where $x_0 = 0$, $x_1 = 1$ and for the other steps $x_i = (x_{i-2} - x_{i-1} \cdot q_{i-2}) \bmod p$. At step i, the devision is performed in the format of $a_i = q_i \cdot b_i + r_i$, where $a_i = b_{i-1}, b_i = r_{i-1}$ and $a_0 = a, b_0 = b$ at step 0. The resulting x will be the equivalence of the inverse of $a \bmod b$.

As an example, assuming that $\delta = 10$, the constant to be hashed $c = 0.6$ and the hash order $p = 17$, then we have

$$Hash(0.6) = \frac{0.6 \times 10}{10} \bmod 17 = 6 \times \frac{1}{10} \bmod 17 \tag{3}$$

Now we use the extended Euclidean algorithm to find the integer equivalence of $\frac{1}{10}$ (the inverse of 10 in $\mathbb{Z}_{17}$). Here, $a_0 = p = 17, b_0 = \delta = 10$. Each step of the algorithm is shown below to calculate x_i, the value of x at that step.

Step 0: $17 = 1(10) + 7$	$x_0 = 0$
Step 1: $10 = 1(7) + 3$	$x_1 = 1$
Step 2: $7 = 2(3) + 1$	$x_2 = (x_0 - x_1 \cdot q_0) \bmod p$
	$= (0 - 1 \cdot 1) \bmod 17 = 16$
Step 3: $3 = 3(1) + 0$	$x_3 = (x_1 - x_2 \cdot q_1) \bmod p$
	$= (1 - 16 \cdot 1) \bmod 17 = 2$
Step 4:	$x_4 = (x_2 - x_3 \cdot q_2) \bmod p$
	$= (16 - 2 \cdot 2) \bmod 17 = 12$

The last value for x is 12, meaning that the integer equivalence of the inverse of 10 ($\frac{1}{10}$ in $\mathbb{Z}_{17}$) is 12. A quick check shows that $12 \times 10 \bmod 17 = 120 \bmod 17 = 1$, so this is indeed correct. Substituting 12 in for $\frac{1}{10}$ in equation 3, we have

$$6 \times \frac{1}{10} \bmod 17 = 6 \times 12 \bmod 17 = 72 \bmod 17 = 4$$

So the constant 0.6 hashes to the value 4 in this example.

2.3.3 The Arithmetic Operators

Because the hashing method takes place in a finite field, all of the standard arithmetic methods are easily handled using *modulo arithmetic*. Hashing of these operators is equivalent to evaluating them within the field:

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

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

$$Hash(A \times B) = (A \times B) \bmod p \tag{6}$$

$$Hash(A \div B) = (A \div B) \bmod p \tag{7}$$

where the division hashing follows the rule of the extended Euclidean algorithm discussed above.

2.3.4 The `if<0` operator

The `if<0` 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. Additionally, it consists of *three* parameters (instead of the usual two): a condition, a true branch and a false branch. All the three parameters must be considered when hashing this operator as well as the order in which they appear. The following approach was formulated to handle this operator:

$$Hash(\texttt{if<0}(A, B, C)) = (\frac{A}{B} + C) \bmod p \tag{8}$$

which uses division and addition to take into account the position of the three parameters.

2.3.5 Operator Closure

All of the functions supported are closed, meaning that for any of the functions $\diamond \in \{+, -, \times, \div, if < 0\}$, $(Hash(A) \diamond Hash(B)) \bmod p = Hash(A \diamond B)$ in $\mathbb{Z}_p$. More specifically:

$$Hash(A + B) = (Hash(A) + Hash(B)) \bmod p \tag{9}$$

$$Hash(A - B) = (Hash(A) - Hash(B)) \bmod p \tag{10}$$

$$Hash(A \times B) = (Hash(A) \times Hash(B)) \bmod p \tag{11}$$

$$Hash(A \div B) = (Hash(A) \div Hash(B)) \bmod p \tag{12}$$

$$Hash(\text{if<0}(A \diamond B, C, D)) = (\frac{Hash(A \diamond B)}{Hash(C)} + Hash(D)) \bmod p \tag{13}$$

This means that by storing already calculated hash values within the tree node structure, one does not need to recalculate the hash values of subtrees each time a tree is to be hashed, as hash values of subtrees can be combined to give correct hash values of the whole tree.

2.4 An Example

Now, we use an example to show the simplification process for a given genetic program. The example genetic 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.

Assume that the hashing order is `17`, `f0` and `f1` are "randomly" assigned the values `3` and `5` respectively. Also, for presentation convenience, Table 2 reiterates the rules in the ruleset that are specifically used in this example.

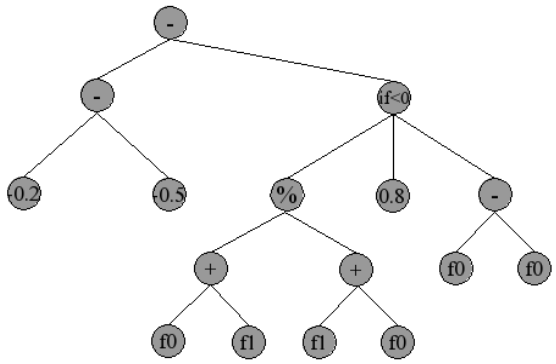


Figure 1: The original program tree.

Table 2: Simplification rules used in this example.

Precondition		Effective Result
(1) `(- a b)`	→	`c`, `c`$= a - b$
(2) `(% A A)`	→	`1`
(3) `(- A A)`	→	`0`
(4) `(if<0 a B C)`	→	`C` (if `a` ≥ 0)

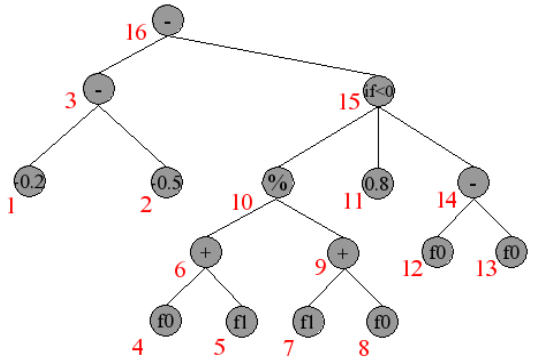


Figure 2: Bottom-up traversal order (shown by integer values).

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 in figure 2.

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 (1) `(- a b)`. The system applies this rule, replacing the subtree with the rule's effective result: "`0.3`".

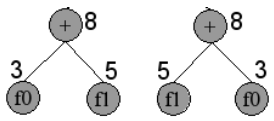


Figure 3: Hashing of two subtrees with same value (shown by integer values).

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 3). Therefore, when node *10* ("`%`") is inspected, the subtree `(% (+ f0 f1) (+ f1 f0))` does indeed match the precondition for rule (2) `(% A A)`. The entire subtree is replaced using the rule to a single node `1`. Similarly, the subtree `(- f0 f0)` matches rule (3) `(- A A)` and is replaced by the single node `0` when the algorithm processes "`-`".

Figure 4 shows the tree after processing nodes *1* through *14*.

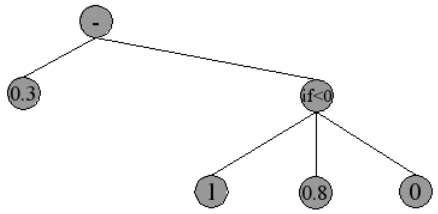


Figure 4: The program after partial processing.

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 (4) `(if<0 c A B)`, 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 (1) `(- a b)`. Applying it yields the final result, a single numerical constant node "`0.3`" (figure 5).

Figure 5: The final program, a single node.

2.5 Summary of the Simplification Algorithm

The simplification algorithm simplifies a given program in the following way:

- Traverse the program tree in a bottom-up fashion using a postfix mode.
- For the terminal nodes, calculate the equivalence hash values.
- For each non-terminal node
  - Calculate the equivalence hash value of the node, directly using the hash values of its child nodes.
  - Iterate through the set of rules. If the node (and its surrounding nodes) match the precondition of a rule, apply that rule to simplify the sub-tree associated with the node.
- Once all nodes have been processed, output the final, simplified program tree.

This algorithm is invoked on a number of programs in the GP system, replacing the original programs with their simplified counterparts.

3 Experimentation Setup

3.1 Datasets

Four datasets, two for symbolic regression tasks and two for multi-class object classification tasks, were used to examine the simplification method. The two symbolic regression tasks represent different "difficulties". The first consists of 200 data points conforming to a simple parabolic curve. The second is a much more complicated piecewise function. We also used 200 data points in [-10, 10] as the fitness cases in this task. An example data set for each task is shown in figure 6.

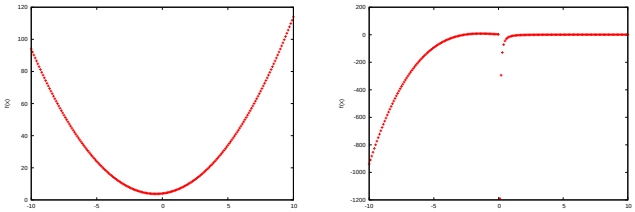
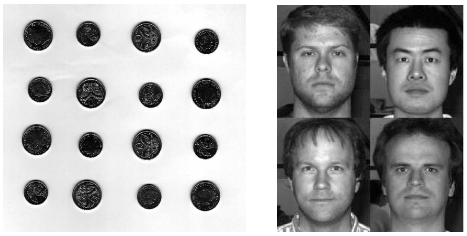


Figure 6: Plots of the two regression task "ideal" solutions.

The first classification task uses a coin dataset. The dataset consists of 480 70x70 pixel image cutouts of New Zealand 5 and 10 cent coins with different sides up and different orientations. These make up four distinct classes: 5 cent heads, 5 cent tails, 10 cent heads, 10 cent tails.

The second classification task uses a subset of the Yale Database B Face Dataset [6]. 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. Example data sets for the two classification tasks are shown in figure 7.



(a) (b)

Figure 7: Object classification data sets. (a) Coins; (b) Faces.

3.2 Terminal and Function Sets

The terminal set consists of several randomly generated *constant terminals* as well as a number of *feature terminals*. The constant terminals are simply n floating point numbers generated in the range of [-1, 1] using a uniform distribution random number generator $(r_0, r_1, ..., r_n)$. The number of feature terminals (m) are task dependent. For the two symbolic regressions tasks, the feature terminal corresponds to the single independent variable. In the coins classification task, there are 8 feature terminals representing the extracted pixel statistic features. In the face dataset, we use 18 feature terminals representing the extracted pixel statistic features from the various facial regions.

$$Terminal\ Set = \{r_0, r_1, ..., r_n, f_0, ..., f_m\}$$

The function set used for these tasks consists of the four basic arithmetic operators, as well as a conditional `if<0` operator. The division used is the commonly used "protected division" where a divide by zero results in zero, removing the *undefined* case. The conditional `if<0` operator takes three parameters, a *condition* which will be evaluated, a *true branch* if the condition evaluates to $<$ zero and a *false branch* if the condition evaluates to $\geq$ zero.

$$\textit{Function Set} = \{+, -, *, \%, \texttt{if<0}\}$$

3.3 Fitness Function

For each 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 patterns of each data set.

For the classification tasks, the fitness of a program is governed by the accuracy of classification, that is, the number of instances correctly classified by the program as a percentage of the total number of instances in the training set. In this approach, we used the static-boundary method (first described in [16]) to translate the single output of a genetic program with a training pattern to a set of class labels. While other methods exist [17, 11], determining the best class translation rule is beyond the scope of this paper.

3.4 Experiment Configuration

Table 3 shows the common parameter values used in the standard GP system and the new GP system with simplification.

Table 3: Genetic programming system parameters.

Task	Gens	Pop.Size	Mut.	Elit.	Cross.	Max.Dep.
Reg_1	50	500	30%	10%	60%	6
Reg_2	50	500	30%	10%	60%	8
Coins	50	500	30%	10%	60%	6
Face	50	500	30%	10%	60%	8

In the new simplified GP, we also used additional "simplification parameters". As mentioned earlier, *hash order* refers to the number of distinct hash values that programs can be hashed to, and *constant precision* is the number of decimal places that are kept when hashing a floating point number. *Proportion* is the percentage of programs in a population to be applied to simplification. Measured in number of generations, *frequency* refers to how often the simplification process is applied. The parameter values used in the experiments are shown in table 4. Frequency at every 0 generation means that the standard GP without simplification is applied.

Table 4: Algebraic simplification: simplification parameters.

Parameter	Value
Hash order (p)	1000077157
Constant precision (δ)	1000000
Proportion	100%
Frequency	Every 0, 1, 2, 4, or 6 gens.

For the coin classification problem, we equally split the data sets into a training set, a validation set and a test set. For the face data set, due to a relatively small number of examples, we used a 10-fold cross validation technique.

Both GP systems run 50 generations for all the four data sets unless it found a solution, in which case the evolution was terminated early. For the coin classification problem, the evolution was also terminated when the accuracy on the validation set started falling down.

All single experiments were repeated 50 runs and the means and standard deviations of the results are presented in the next section.

4 Results and Discussion

Table 5 shows results of the two GP approaches on the four data sets in terms of the effectiveness (best fitness — mean squared error for regression and classification accuracy for classification), training efficiency (number of generations and training time), and average size of all the programs in the systems in number of nodes.

Table 5: Results for each of the four tasks.

Task	Frequency	Best Fitness	Generations	Time(s)	Avg. Prog Size
Reg_1	without	0.005 ± 0.013	28.781 ± 13.427	1.221 ± 0.501	37.611 ± 5.634
	Every 1	0.011 ± 0.042	32.438 ± 13.119	1.232 ± 0.464	25.606 ± 2.937
	Every 2	0.005 ± 0.013	31.562 ± 14.291	1.109 ± 0.486	27.232 ± 3.667
	Every 4	0.027 ± 0.119	31.094 ± 13.359	1.071 ± 0.494	28.412 ± 5.074
	Every 6	0.003 ± 0.009	31.688 ± 12.228	1.070 ± 0.359	29.200 ± 4.581
Reg_2	Without	83.774 ± 75.283	44.875 ± 4.756	5.141 ± 1.019	104.436 ± 22.171
	Every 1	92.884 ± 80.624	44.875 ± 4.756	5.206 ± 0.861	74.362 ± 13.642
	Every 2	67.346 ± 59.315	44.875 ± 4.756	4.270 ± 0.759	74.841 ± 13.886
	Every 4	82.471 ± 85.606	44.875 ± 4.756	4.152 ± 1.069	77.337 ± 21.487
	Every 6	85.301 ± 93.883	44.875 ± 4.756	3.989 ± 0.627	75.549 ± 12.840
Coins	Without	0.973 ± 0.025	35.750 ± 11.200	1.657 ± 0.532	44.476 ± 7.302
	Every 1	0.964 ± 0.039	37.469 ± 10.992	1.700 ± 0.452	32.539 ± 5.622
	Every 2	0.974 ± 0.028	35.031 ± 11.290	1.492 ± 0.407	34.720 ± 4.253
	Every 4	0.974 ± 0.032	36.656 ± 10.527	1.477 ± 0.411	34.884 ± 5.264
	Every 6	0.954 ± 0.054	37.250 ± 10.336	1.522 ± 0.355	36.566 ± 3.919
Faces	Without	0.855 ± 0.117	46.077 ± 3.578	2.646 ± 0.578	37.861 ± 8.755
	Every 1	0.876 ± 0.104	45.712 ± 4.415	2.622 ± 0.583	29.798 ± 6.571
	Every 2	0.867 ± 0.117	45.885 ± 3.717	2.367 ± 0.460	29.364 ± 5.966
	Every 4	0.851 ± 0.105	46.077 ± 3.578	2.251 ± 0.441	28.917 ± 5.757
	Every 6	0.866 ± 0.075	46.077 ± 3.578	2.288 ± 0.464	30.081 ± 6.274

4.1 Effectiveness

As can be seen from table 5, the GP approach with the proposed simplification at different frequencies almost always achieved comparable or even superior fitness, either 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 classification 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 and the GP evolution is difficult to make further improvement since the crossover operator starts to destroy good existing building blocks, the simplification algorithm actually

generates new genetic materials which might contain new good building blocks by *reorganising* the entire genetic programs. This makes it possible to consider the simplification as a new genetic operator in the future.

We also observe that the GP approach with simplification in every generation achieved worse performance than every two to six generations in most cases. Although this simplification could introduce new genetic materials like mutation, the programs will not have sufficient chances for evolution if we apply the simplification too often. Applying simplification less frequently will give the GP system more chances to perform evolution than that in every generation.

4.2 Efficiency

According to Table 5, while the numbers of generations used for the evolutionary training process for different GP systems were fairly similar, the actual training CPU times are quite different. As expected, the GP approach with the simplification almost always improved the training efficiency and in some cases very much so. This is mainly because the simplification process removes the redundancy, makes the genetic programs shorter, and accordingly reduces the search space.

Not surprisingly, the GP system with simplification at every generation generally led to a very slight increase in training time in most cases (except for the face data set where there was a slight improvement). This can be attributed to the overhead introduced into the system by the simplification component. This overhead, when occurring at every generation, usually outweighs the time saved from processing smaller simplified programs.

4.3 Program Size

As can be seen from the last column of table 5, the average size of the programs is significantly reduced for the GP system with simplification at all frequencies 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.

To do a further analysis, we also present the average size of genetic programs at every generation for the four tasks in figure 8. While performing simplification at lower frequencies results in a higher average program size than performing at every generation, this increase in size is very small, suggesting that simplification does not need to be performed at every generation.

4.4 Simplification Frequency Analysis

In most data sets, applying simplification at every generation led to a slight loss in fitness and a slightly higher computational cost. This suggests that in general, simplification should not be applied to evolution at every generation.

While GP with different simplification frequencies results in different results, it is always possible to find a good one that can achieve better effectiveness and better efficiency than the basic GP without simplification. However, this is generally task dependent and usually needs an empirical search. But if such a search can improve the system performance significantly, this is a small price to pay. Our experiments suggest that simplification at every 2 generations could serve as a starting point.

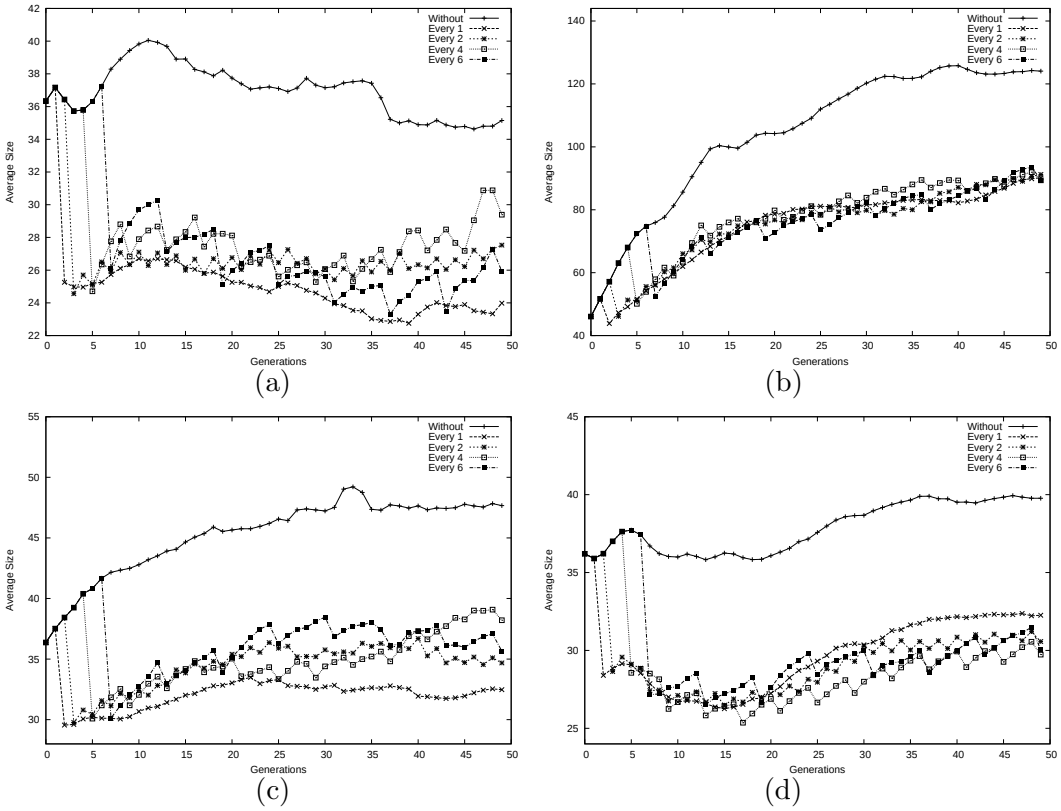


Figure 8: Average program size per generation. (a) Reg_1; (b) Reg_2; (c) Coins; (d) Faces.

5 Conclusions

The goal of this paper was 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 an bottom-up fashion by an postfix order, and applying the simplification rules along with an algebraic equivalence component to non-terminal nodes to simplify programs directly without needing to translate it into another format.

The GP system with the simplification algorithm was examined and compared with the basic GP approach without simplification on four regression and classification problems of varying difficulty. The results suggest that, the new simplification approach outperformed the basic GP approach in terms of effectiveness, efficiency and program size on these data sets.

The results also suggest that performing simplification at every generation is not recommended and simplification at every two generations could serve as a starting point.

The online simplification during evolution seems to be able to reduce the search space. While it could introduce new genetic materials, 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.

In the current approach, we applied the simplification to all individual programs in the population. We will investigate whether the performance can be further improved if we only simplify a proportion of programs in the population. We will also investigate what effects would be produced if we consider the simplification as a new operator and put it into the function set.

6 Acknowledgement

This work was supported in part by the Marsden Fund of New Zealand under grant No.05-VUW-017 and the University Research Fund 06/9 at Victoria University of Wellington.

References

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*. Morgan Kaufmann Publishers, 1998.

[2] T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation*, pages 33–38, Germany, 1994.

[3] J. Busch, J. Ziegler, C. Aue, A. Ross, D. Sawitzki, and W. Banzhaf. Automatic generation of control programs for walking robots using genetic programming. In *EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming*, pages 258–267, London, UK, 2002.

[4] B. Cherowitzo, 2006. Lecture Notes. http://www-math.cudenver.edu/ wcherowi/courses/m5410/ exeucalg.html. Visited on 7 January 2006.

[5] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[6] A. Georghiades, P. Belhumeur, and D. Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 23(6):643–660, 2001.

[7] G. H. Gonnet. Determining equivalence of expressions in random polynomial time. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 334–341, New York, USA, 1984.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[9] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[10] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, New York, NY, USA, 1986.

[11] T. Loveard and V. Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077, Seoul, Korea, 2001. IEEE Press.

[12] W. A. Martin. Determining the equivalence of algebraic expressions by hash coding. *j-J-ACM*, 18(4):549–558, Oct 1971.

[13] R. Poli. Genetic programming for image analysis. In *Proceedings of the First Annual Conference*, pages 363–368, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[14] T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In *Proceedings of the First Annual Conference*, pages 215–223, Stanford University, CA, USA, 28–31 1996. MIT Press.

[15] W. Trappe and L. C. Washington. *Introduction to Cryptograpy with Coding theory.* Prentice-Hall, 2ed edition, 2006.

[16] M. Zhang and V. Ciesielski. Genetic programming for multiple class object detection. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence*, volume 1747 of *LNAI*, pages 180–192, Sydney, Australia. 1999. Springer-Verlag.

[17] M. Zhang and W. Smart. Multiclass object classification using genetic programming. In *Applications of Evolutionary Computing, EvoWorkshops2004*, volume 3005 of *LNCS*, pages 369–378, Portugal. 2004. Springer Verlag.