Discovery of Understandable Math Formulas Using Genetic Programming

Timothy Lai
Computer Science Program
Stanford University
Stanford, CA 94305
http://www.stanford.edu/~timlai
timlaip@hotmail.com

ABSTRACT

Genetic programming (GP) can be applied to a wide variety of problems and produce human competitive results, but the solution GP comes up with is often hard to understand. This paper shows that picking the right functions sets, setting up the appropriate program structure, and adding a parsimony factor can help to reduce the complexity of the evolved solution and to make the evolved solution easier to understand. Even though the algorithm for finding the greatest common factor (GCF) of two positive integers is well known, this paper will use evolving the GCF algorithm as an example and show that the right program structure, functions sets, and an appropriate parsimony factor can reduce computation time, decrease solution size, increase the understandability of the evolved program, and make the evolved algorithm more generalizable.

1. Introduction

Genetic Programming has been used in a variety of fields to solve difficult problems and produce human competitive results. In the field of mathematics, it has been used to solve problems such as symbolic regression, discovery of trigonometry identities, and sequence induction, just to name a few (Koza 1992). However, in many cases, even though genetic programming correctly solves the problem, the algorithm that genetic programming comes up with is large, complex, and difficult to understand. This paper will aim to find the best strategies and structures to use in order for genetic programming to produce a result that is more understandable. The problem of finding the greatest common factor of two positive integers will be explored. Since the GCF problem, like many other math problems, has an infinite number of test cases, increasing the understandability of the evolved solution also gives us one additional way to validate the correctness of the evolved program. Upon the successful discovery of a solution to solve the GCF problem, additional adjustments to the GP run such as modifying the function set, changing the program tree structure, and adding additional fitness measures will be tried to see if we can make GP produce a solution with an understandable algorithm.

2. Background of GCF Problem

The greatest common factor of two positive integers, A and B, is defined as the largest integer that divides both A and B evenly. The solution to finding the greatest common factor of two positive integers is well known. One simple approach is to iterate through all the integers between 1 and the smaller of the two integers A and B, and the largest integer that divides evenly into both A and B is the GCF. A more efficient approach can be obtained from the following math theorem (Rosen 1995):

Let A and B be two positive integers with A >= B.
If C = A mod B is non-zero, then GCF of A and B is equal to GCF of B and C.
Otherwise, B is the GCF of A and B.

Applying the above theorem, the following is a pseudo code representation of the Euclidean algorithm that can be used to solve the problem of GCF.

```
// returns GCF of a and b
int gcf(int a, int b)
{
   int rem = a mod b;
   while (rem != 0)
   {
      a = b;       // updating dividend
      b = rem;     // updating divisor
      rem = a mod b;
   }
   return b;
}
```

3. Genetic Programming Setup

3.1 Major Preparatory Steps

Table 1 shows the setup for the GP run using iteration and memory to evolve an algorithm for the GCF problem. This setup is aimed to produce small and understandable results.

Objective	The objective is to evolve a program to find the greatest common factor of two positive integers.
Terminal set	The program is separated into two branches. One initialization branch and one for-loop branch. Both branches share the same terminal set: {T1, T2, T3, T4, X, Y, Zero, One}
Function set	The function set for the initialization branch is: {Prog1, Prog2, SetT1, SetT2, SetT3, SetT4} The function set for the for-loop branch is: {Prog1, Prog2, SetT1, SetT2, SetT3, SetT4, +, -, *, %, Mod, If, Not, Equal}
Fitness cases	12 fitness cases: GCF(125, 15) = 5 GCF(36, 24) = 12 GCF(5, 1) = 1 GCF(36, 25) = 1 GCF(18, 14) = 2 GCF(16, 12) = 4 GCF(65, 25) = 5 GCF(65, 13) = 13 GCF(24, 15) = 3 GCF(100, 10) = 10 GCF(180, 72) = 36 GCF(27, 27) = 27
Raw fitness	Raw fitness is the sum of the following two components: ValErr = Sum of the absolute value of the difference between the actual GCF and the calculated result. ValSize = Number of nodes in the initialization branch and for-loop branch Raw fitness = ValErr + ValSize / weight, where weight = 100
Standardized fitness	Same as raw fitness.

Hits	Number of test fitness cases gotten correctly. Max is 12.
Wrapper	None
Parameters	Population size M = 1500 Number of generations G = 150 Reproduction rate = 0.1 Crossover rate = 0.9 Tournament selection of size 7 is used.
Success predicate	When number of hits equals 12, additional test cases are used to test the success of the program. However, no termination criteria is used in hope that the size fitness factor will drive the evolved program to be even smaller.

Table 1 Genetic programming tableau for GCF problem

3.2 Description of Terminals and Methods

Terminals

T1, T2, T3, T4: These four variables represent the memory block of the run. Each terminal Tn is an integer variable that can be set and accessed during the run.

X: X is the first of the 2 positive integers whose GCF we are trying to find. We make X the greater of the 2 integers.

Y: Y is the second of the 2 positive integers whose GCF we are trying to find. Y is the smaller of the 2 values.

Zero: Constant 0.

One: Constant 1.

Methods

Prog2, Prog3: These are connectors in the tree that allow the children nodes to be executed sequentially. ProgN has N children.

+, -, *, %: These are the add, subtract, multiply, and protected divide arithmetic operators. Each takes 2 arguments.

Mod: This is the mod operator. It takes two arguments *arg1* and *arg2*, and returns the value *arg1* mod *arg2*. If *arg2* evaluates to 0, then the return value is 1.

If: The If operator is the conditional operator that takes 3 arguments. The first is the expression to be tested. If the first argument evaluates to a non-zero value, then the second child is executed. Otherwise the third child is executed.

Equal: The Equal operator takes 2 arguments and returns 1 if both children evaluate to the same value. Otherwise it returns 0.

Not: The Not operator takes 1 argument and returns 1 if the child argument evaluates to 0. It returns 0 otherwise.

3.3 Tree Structure

Two branches of trees are evolved during the run. The first branch is an initialization branch that takes care of setting the variable values before the execution of the second branch. The initialization branch contains all the terminals, the SetTn() variable modifiers, and the connectors Prog2 and Prog3.

The second branch is the main for-loop branch representing the body of the for-loop. The terminal T1 is arbitrarily chosen to be the return value of the evolved individual upon the completion of the for-loop. T2 is the variable representing the for-loop termination criteria. At the beginning of each iteration, the value of T2 is checked. If T2 is equal to zero, then we break out of the for-loop. The for-loop is hard coded to iterate a maximum of 50 times to avoid infinite loops.

The following pseudo code may help to understand the evaluation of the two branches better.

```
// global vars T1, T2, T3, T4
Init(); // initialize T1, T2, T3, T4 to zero
EvaulateTree (Initialization Branch);
For (int j=0; j<50; j++)
{
   if (T2 == 0)
      break;

   EvaluateTree (For-loop Branch);
}
// value of T1 at the end of evaluation is taken to be the result
```

3.4 Fitness Evaluation

3.4.1 Training Test Cases

Twelve fitness cases were chosen for the run. These twelve cases were not chosen at random because the method of generating two random positive integers to use seems to favor training cases where the GCF is composed of multiples of relatively small prime numbers. Choosing from a large range of integers doesn't seem to gain much advantage in obtaining a representative training set. Having multiple test cases helps to train the program correctly to achieve a general solution, but it is desirable to keep the number of test cases small to speed up the genetic programming run.

Because of the above reasons, twelve fitness cases were chosen by hand. These fitness cases include general test cases where the two integers share many common factors, where the two integers share only a few common factors, and where they share only one prime factor. Special care is given in choosing these general test cases so that the number of iterations that the Euclidean algorithm needs to find the GCF of the two numbers varies. Corner test cases where the two integers are relatively prime, where the two integers are the same, and where the smaller integer is the GCF of the two integers are also included in the training set.

3.4.2 Hits

The hits in the run is defined to be the number of fitness cases gotten correctly. A maximum score of 12 hits is obtainable. However, scoring perfect on the 12 test cases doesn't guarantee a 100% correct general solution. Therefore when an individual scoring 12 hits is found, additional test cases involving bigger numbers and test cases that require more iterations to solve are used. Early on in the experiment, the genetic programming run is terminated when the individual scores a perfect score on the additional test cases. But in later runs, this termination criteria is taken out in hope that the parsimony factor will help to reduce the size of the evolved program even more to make the program more efficient and readable.

3.4.3 Raw and Standardized Fitness

Raw fitness was originally defined to be the sum of the absolute value of the differences between the actually GCF and the calculated results. But another component, the weighted sum of number of nodes in the initialization branch and the for-loop branch is added later. Overall the raw fitness = #hits + #nodes/weight. In the discussion that follows, the weight used during the run is 100 unless otherwise specified. The standardized fitness is the same as the raw fitness. Adjusted fitness is defined as 1 / (1 + Raw Fitness) to ensure that it is in the range between 0 and 1.

3.5 Example Individuals

In figure 1 below, the initialization branch consists of the 2 nested functions SetT2 and SetT3. Structures of this kind turn out to be common in the initialization branch. In the initialization branch of this example individual, T3 is set to X, and SetT2 takes the return value of SetT3(X), which is X again, as an argument and sets T2 to X also. The main for-loop branch consists of an If-statement that checks if T3 mod Y is equal to zero. If so, T1 is set to Y, and T2 is set to zero, thus terminating the for-loop. If not, T4 and T1 are both set to one, and this step is repeated 50 times in the for-loop before the iteration limit is reached. The effect of this individual is that if Y is the GCF of X, then it correctly returns Y as the result. Otherwise it returns 1 as the GCF of X and Y.

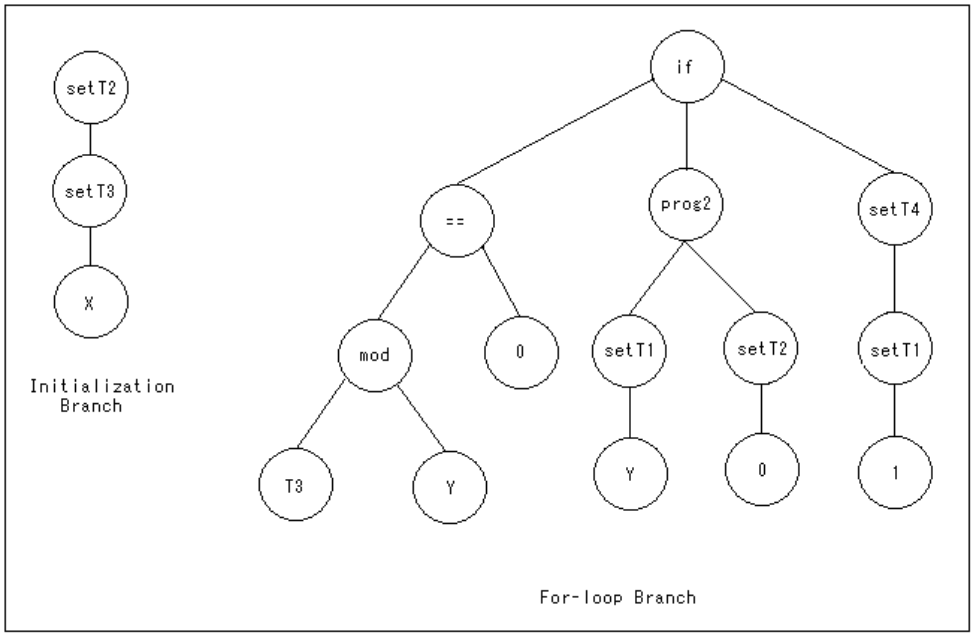

Figure 1 An example of a possible individual in the population.

Figure 2 below shows a 100% correct solution. In the initialization branch, T1 is set to Y, T3 is set to X, and T2 is set to X mod Y. The body of the for-loop consists of three operations connected by Prog3. The body of the for-loop is only evaluated if T2 is not equal to zero. If T2 is not equal to zero, we move the divisor to T3, and remainder to T1, and set T2 to be T3 mod T1. If the new remainder is not zero again, this procedure repeats in the for-loop. This simulates the Euclidean algorithm.

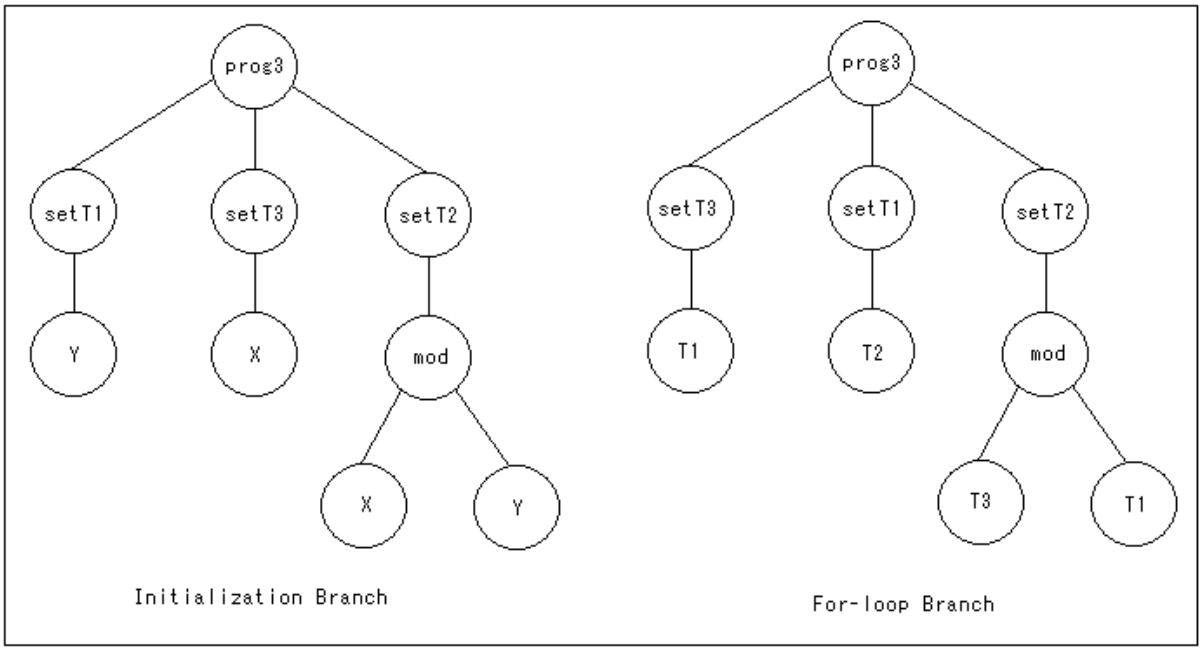

Figure 2 An example of a perfect individual structure in the population.

4. Problems Encountered

The problems described below are from earlier experiments in which only one result producing branch is evolved during the run.

4.1 Do-Until Operator

In earlier experiments of GP runs, the structure of having two branches in the tree wasn't used. Instead, a Do-Until (DU) operator was used in the function set along with the other ones described in section 3.2. The DU operator takes two arguments. The first argument serves as the termination criteria, and the second argument is the body of the loop. While the first argument evaluates to true (a non-zero value), the second argument is evaluated. A maximum of 50 iterations is also enforced to prevent infinite loops.

By putting the DU operator in the function set in a GP run that produces only one result producing branch, we hoped that GP will evolve both the program structure and the algorithm needed to solve the GCF problem. Indeed, some individuals achieving 12 hits (perfect for the number of training cases presented) was evolved; however, there are many problems with this approach in terms of time, complexity, program size, and understandability of the evolved program. With the DU operator, a generation of 1000 individual may take up to 30 minutes to process. The time it takes to evolve each generation tend to increase during the course of the run as the individuals in the population increase in size. Another reason contributing to the large amount of time needed to evolve one generation is that multiple DU operators may exist or be nested in the program. In many cases, the DU operators contain termination criteria that can never be satisfied. In other cases the DU operators contain bodies that do redundant work such as setting the same variable to the same value repeatedly. These unnecessary DU operations reach their max iteration limit after 50 loops, and the produced effect can easily be replaced with a single SetTn() operator. Nested DU operators pose even greater problems because the number of iterations the inner-most DU body gets executed grows exponentially large at the rate of 50^n, where n is the number of nested DU operators. The possible appearance of multiple DU operators make the program longer to run, harder to understand, and unnecessarily complex.

4.2 Lack of Generality

Without constraints in size, these programs also seem to grow larger and larger in order to fit the 12 test cases. But the results are often not general. The programs seem to evolve more complex in order to find a way to solve the 12 training cases. specifically Instead of using the DU operator to iteratively test the remainder of the division of two numbers, and then updating the dividend and divisor appropriately as in the Euclidean algorithm, the evolved program tend to mimic this iterative solution as a bunch of If-Else statements that carries the generalized algorithm out to only a fixed number of iterations. For example, instead of using the terminals Tn as place holders for divisors and dividends and updating them correctly, some of the programs contains code fragments like (% (% X Y) Y) to simulate the second iterative step of the Euclidean algorithm. These individuals score relatively high and survive because they can correctly solve all test cases requiring only a pre-specified number of iterations of the Euclidean algorithm.

Multiple approaches were considered to help GP evolve a generalized solution. One approach is to use random test cases in each generation. However, this approach was ruled out because of the difficulty in generating good random test cases as discussed earlier in section 3.4.1. Another approach is to increase the number of test cases to add more diversity to the training set. This is a good idea although adding test cases will lead to even slower runs. So 8 additional test cases involving larger numbers and larger number of iterations to solve were added. To save time, these 8 test cases will only be tried on the best-of-generation individuals if they have 12 hits already. The old termination criteria of stopping the run after achieving 12 hits is replaced by terminating only after the additional 8 test cases are satisfied also. These test cases can serve as negative testing in case the perceived perfect solution turns out to be flawed on some unseen sample data. They add more assurance that the evolved program is correct and generalizable, but passing these 8 test cases still cannot guarantee the evolved program's general applicability.

4.3 New Program Structure Needed to Save Computation Time

To deal with the problems of the computation time needed to do the GP run and the complexity of the evolved solution, the idea to modify the structural representation and evolve an initialization branch and a separate main for-loop branch came about. This new tree structure guarantees the existence of only one and no nested loops in the evolved program to help with 1) speeding up the run, 2) making the program more understandable by separating it into two specialized branches, and 3) getting rid of unnecessary functions and terminals in certain branches to reduce unnecessary choices.

5. Results

In generation 55 of one run, a perfect individual is evolved. This individual scores a perfect 12 on the training test cases and also has a perfect score on the additional test cases. The initialization branch contains 3 nodes, and the main for-loop branch contains 14 nodes as shown below:

```
INIT:
```

```
   (setT2 (setT3 X))
  FOR_LOOP:
   (setT1 (if (% t3
                 (setT1 (if t1 t1 Y)))
              (% t3
                 (setT3 t1)) t1))
```

This was run with a population size of 1500 individuals with reproduction rate of 10% and crossover rate of 90%, using tournament-selection of size 7. A parsimony factor is included as part of the fitness measure, and the weight of the parsimony factor is 100. The choice of 1500 as the population size is carried over from the older GP runs of one branch only. 1500 seems to be the smallest population size for which GP can still evolve a solution with one result-producing branch only, so this number is used here again for comparison purposes.

Upon closer inspection, this individual mimics the Euclidean algorithm described earlier. In the initialization branch, T3 and T2 are both set to X. T3 acts as the dividend place holder. T2 is initialized to a non-zero value X so that the for-loop doesn't terminate prematurely. In the first iteration, the inner If-statement evaluates to false since T1 is initialized to zero; therefore, the inner SetT1 operator sets the value of T1 to Y. Then the outer If-statement tests to see if T3 mod T1 is equal to zero. If so the GCF of X and Y has been found already and is the value currently stored in T1. In this case, the dummy operation of setting T1 to the value of T1 is performed repeated until the maximum number of iteration is reached. If T3 mod T1 is not equal to zero, then the divisor place holder T1 is set to the value of T3 mod T1, and the dividend place holder T3 is updated to be the value of the previous divisor. This is essentially the logic behind the Euclidean algorithm.

In fact, during this run, an individual scoring 12 hits was already obtained in generation 19 with the following tree:

```
INIT:
 (setT2 (prog3 (setT4 (prog3 Y
                              (setT4 X) X))
               (setT3 (setT2 (setT1 (setT4 (setT2 X))))) t3))
FOR_LOOP:
 (setT1 (if (% t3
               (setT1 (if (setT1 (% (setT4 t1) Y))
                          (setT4 t1) Y)))
            (% t3
               (setT1 (if (setT3 Y)
                          (setT4 t1)
                          (setT2 t2))))
            (if (prog2 t4
                       (setT4 t1))
                (setT4 t1)
                (setT2 t2))))
```

Even though the above individual is already a lot more compact than the ones evolved in the runs where there is no parsimony factor, this individual still contains operations that essentially performs nothing or can be replaced by simpler ones. Keeping the run going, in generation 29, the following best-of-run individual is reduced to 29 nodes as shown below:

```
INIT:
 (setT2 (setT3 X))
FOR_LOOP:
 (setT1 (if (% t3
               (setT1 (if t1
                          (setT4 t1) Y)))
            (% t3
               (if (setT3 Y)
                   (setT4 t1)
                   (setT2 t2)))
```

Notice that this individual already resembles the final result very closely and in fact contains the same underlying algorithm with a few redundant method calls. In particular, the initialization branch is already the same as the one in the best-of-run individual. The main for-loop branch has very similar structure as the best-of-run individual but contains some unnecessary calls such as setting the otherwise unused T4 variable. In generations 33, 34 and 36, this best-of-generation individual's for-

loop branch kept decreasing by one in size. The unnecessary calls to SetT4 are either eliminated from the tree or replaced by the return value of the SetT4 call only.

In generation 43, the best-of-generation individual is almost the same as the final result.

```
INIT:
 (setT2 (setT3 X))
FOR_LOOP:
 (setT1 (if (% t3
               (setT1 (if t1 t1 Y)))
            (% t3
              (if (setT3 Y) t1 t1)) t1))
```

This individual has the program structure to run Euclidean algorithm. The only thing wrong with it is that it is updating the dividend with the constant and incorrect value of Y every time. Upon closer inspection, fixing the dividend value in the algorithm has the same effect as specifying a fix number of iterations in the Euclidean algorithm.

Finally in generation 55, the assignment of the dividend variable is updated correctly and the Euclidean algorithm is evolved. The individual in generation 55 actually doesn't include a correct termination condition unless Y happens to be the GCF of X and Y. Instead, the evolved program stops updating the result variable T1 when the GCF is found and does the dummy operation of setting T1 to be T1 for the remaining of the iterations. Although not perfect in terms of its termination condition, this individual takes advantage of the way for-loops are defined in the GP run and is perfect within the framework of our setup.

6. Result Discussion

Before including a parsimony factor, the programs that the GP runs produced were large and extremely hard to understand. In order to better understand the evolved programs, a parsimony factor is used, and the fitness measure became a combination of the error from the actual result and the size of the evolved tree. The effect of the parsimony factor was satisfying. The evolved programs reduced in size dramatically, and as a side effect of keeping the run going and pressuring the ideal individuals to get smaller, the evolved programs also tend to generalize better.

6.1 Size Comparison without Parsimony

When we first moved to the two branches structure, a parsimony factor was not used, and the evolved solutions were large in comparison to the individual shown in section 5. The parsimony factor reduced the program size dramatically. As an example, one perfect individual from a GP run without parsimony factor has 128 nodes with the structure below:

```
INIT:
 (setT1 (prog3 (setT2 (setT1 X))
               (setT4 (setT2 (setT1 (setT2 (setT4 (setT2 (setT2 X)))))))
               (setT4 (setT2 (setT2 (setT1 (setT2 (setT1 X))))))))
FOR_LOOP:
 (if (setT3 (prog2 (+ (prog2 (setT1 (== Y X))
                             (setT1 (== Y X)))
                      (== 1 t4))
                   (setT1 (== Y X))))
     (* (+ (* (prog3 t3 t3 t3)
              (prog2 (== 1 t3)
                     (setT4 t4)))
           (not (setT1 Y)))
        (prog4 t3
               (== (setT3 t3)
                   (if X 0
                       (- 1 0))) 0
               (setT3 (% t1
                         (prog4 (setT2 (setT1 t4))
                                (if X 0 t3)
                                (setT3 t3)
                                (setT3 (% t1 Y)))))))
     (prog4 (+ (* X Y)
```

```
   (== 1 t4))
(not (% (+ (setT1 1)
           (- 1 0))
         (setT4 t2)))
(setT4 (setT2 (setT1 t4)))
(setT2 (% (setT1 (prog4 (setT2 (setT3 (% t1 Y)))
                        (if X 0 t3)
                        (setT3 t3)
                        (- Y t2)))
          (setT2 (setT1 t4))))))
```

This individual also scores perfectly on the additional test cases, but it is unknown whether this is truly a 100% correct solution because of its complex structure. The algorithm this individual uses is almost impossible to understand.

In contrast, the following perfect individual with only 10 nodes is evolved with the help of the parsimony factor and no termination criteria in generation 38 of one GP run. This individual follows the Euclidean algorithm, including the termination condition, perfectly.

```
INIT:
 (prog2 (setT2 Y)
       (setT1 X))
FOR_LOOP:
 (setT2 (% t1
         (setT1 t2)))
```

Without the parsimony fitness measure, the average size of 9 individuals scoring 12 hits is 100. With parsimony as a fitness measure, the average size of 6 individuals scoring 12 hits is only 19. The average tree size of individuals evolved with and without parsimony as a fitness measure is shown below. This is the average taken over 20 runs each.

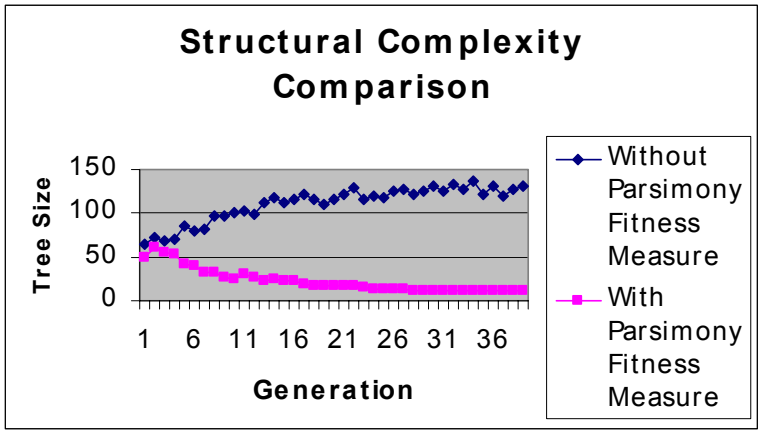

Figure 3 Average individual size with and without parsimony fitness measure.

6.2 Weight of Parsimony Factor

The choice for the weight of the parsimony factor is important. The fitness measure is defined to be raw fitness = error + size/weight, where error is the sum of the absolute value of the difference between the actually GCF and the calculated result. The parsimony factor is the number of nodes in the tree divided by the weight factor. When we overemphasize the parsimony factor, for example making weight equal to one, then the overwhelming advantage for a small program in the evolutionary process causes the accuracy of the program to be overlooked. For example, when weight equal to 1, many small programs of size less than 10 turn out to be the best-of-generation individuals even though they only score between 2 to 4 for the number of hits. For the GCF problem, the parsimony factor has to be properly adjusted so that accuracy on the training test cases is emphasized. With the proper weight for the parsimony factor, the GP run produces correct programs with unnecessary parts stripped off to make the solutions smaller and more understandable. Choosing the weight to be between 100 and 200 works well for this particular setup of the GCF problem.

6.3 Choice of Fitness Cases Important

In evolving the GCF program, the best-of-generation individuals early in the run were often large programs that solve the fitness cases correctly but fail to generalize on additional unseen test cases. One of the problems contributing to this lack of generalization was the bad choice of some fitness cases initially. Five out of the twelve original test cases happen to share the property that the GCF of the two integers is the same as the difference between the two. This confusing characteristic causes many individuals with perfect scores on the training cases to miscalculate the GCF of some unseen test cases to be the difference between the two integers. But even after changing the test cases to cover a wider variety of training cases, the problem of evolving a generalized solution still remains although the solutions evolved through the modified training cases do perform better and solve a larger variety of unseen test cases.

6.4 Parsimony Contributes to Generality

Fortunately the parsimony factor also seems to increase the general applicability of the evolved program in addition to reducing program size and enhancing understandability. Without the parsimony factor, the individuals in the population are free to grow arbitrarily large to cover the twelve training cases. Without the parsimony factor, it seems easier for GP to find solutions that cover all training cases, but the ability of these solutions to generalize is not guaranteed. However, by putting in a parsimony factor, it seems that the evolved programs are forced to find a general strategy to solve all training cases. In 20 runs of GP without the parsimony factor, nine individuals scoring a perfect twelve hits are evolved. However, three out of these nine individuals failed on additional test cases. In 20 runs of GP with the parsimony factor, only six individual scoring twelve hits are evolved. But out of these six, only one failed on additional test cases. From tables 3 and 4 below, smaller programs seem to generalize better.

Size	Additional Cases Correct
168	62.5%
121	100%
113	75%
94	75%
93	100%
82	100%
81	100%
75	100%
74	100%

Table 3 Size and generalizability of 9 individuals scoring 12 hits from 20 runs without parsimony factor.

Size	Additional Cases Correct
38	87.5%
18	100%
17	100%
16	100%
15	100%
10	100%

Table4 Size and generalizability of 6 individuals scoring 12 hits from 20 runs with parsimony factor.

7. Conclusion

The GCF problem is a simple problem for genetic programming to solve given that an appropriate set of test cases is chosen. However, without additional care, the solution that genetic programming comes up with can be hard to understand. This paper has demonstrated that with an appropriate parsimony factor to reduce the size of the evolved solution and a proper choice for the structure of the evolved program, we can make genetic programming produce solutions that are compact, easy to understand, and more general.

8. Further Work

Further experiments using GP with more complex setup like recursion and automatically defined functions can be made to see if they can solve more complex math problems and provide understandable results that give insight to the algorithm discovered.

Bibliography

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

Rosen, Kenneth H. 1995. *Discrete Mathematics and its Applications*. New York NY: McGraw-Hill, Inc.