Using Genetic Programming To Evolve an Algorithm For Factoring Numbers

Jenny Rose Finkel
Computer Science Department
Stanford University
Stanford, CA 94305
`jrfinkel@stanford.edu`

Abstract

This paper demonstrates how genetic programming can be used to evolve an algorithm to correctly factor positive integers.

1   Introduction

Factoring arbitrarily large numbers is a hard problem; there are no knpwn algorithms which solve the problem efficiently. There are, however, lots of inefficient algorithms which correctly factor numbers. This paper demonstrates how to use genetic programming (GP) to evolve an algorithm which takes a positive integer and returns a non-trivial factor of that number. An algorithm $factor(n)$ is a solution to this problem if it returns either 1 or n when n is prime, or an x such that $n \bmod x = 0$ and $x \neq 1$ and $x \neq n$ when n is not prime..

There are many reasons why one would want to find an efficient solution to this problem. Modern cryptography is based on the assumption that factoring numbers takes a long time. However, for now we are only interested in finding a correct, general algorithm. Finding a function which satisfies this requirement is also an interesting problem, because unlike many function approximations there is no such thing as an 'almost right' answer. If an algorithm returns a proposed factor, f, for an integer, n, such that $n \bmod f = 1$, that answer is no better than any other proposed factor such that $n \bmod f! = 0$.

The following section describes the methods used, including the terminals (§2.2), functions (§2.3) and fitness function (§2.5). Parameters are discussed in §3 and results presented in §4. If you wish to skip straight to reading about all of my mistakes, you will find then in §5.

2   Methods

For this project I used Sean Luke's Evolutionary Computation and Genetic Programming Research System (ECJ) which is freely available on the web at http://www.cs.umd.edu/projects/plus/ec/ecj/. The GP used was weakly typed - all terminals and functions return integers.

2.1   Side Effects and the Order of Execution

While some of the code displayed in this paper is printed using a LISP-like notation is important not to view it as pure LISP code. The GP is given two settable variables and functions for setting them. This means that the code has side effects and executing identical pieces of code located at different parts of the program, or even the same piece of code at two different times, may result in different values and different state (the state we are concerned with is the state of the settable variables). In the following example, `set-sv0` takes one arguement and sets the variable `sv0` to that value and is itself equal to that value:

```
(+ (set-sv0 (+ sv0 1)) (set-sv0 (+ sv0 1)))
```

Objective:	Find an algorithm for factoring positive integers
Terminal Set:	x, the number to be factored.
	sv0, a settable value who's initial value is 0.
	sv1, a settable value who's initial value is 1.
Function Set:	set-sv0
(see §2.5 for	set-sv1
further descriptions)	mod
	add
	equals
	less-than
	if-then-else
	do-while
Fitness Cases:	The integers 1 through 100
Hits	The number of integers correctly factored
	However, if x (the number being factored) is not in the
	program, then no credit is given.
Raw Fitness:	two parts added together:
	a) $100 - number\ of\ hits$
	b) a float whose value is less than one and is proportional to
	the size of the GP tree (see §2.5.3 for further description)
Standardized Fitness:	same as raw fitness
Parameters :	Population size M = 5000
	Elitism is used (10 best)
Success Predicate :	The evolved program correctly factors all of the integers
	in the fitness cases and has a size of less than or equal
	to 20 nodes ($raw fitness \leq 0.02$).

Table 1: Genetic programming tableau for number factoring problem.

This code contains the snippet (set-sv0 (+ sv0 1)) twice, but each evaluate to a different value (specifically, the value of the second occurance is one more than the value of the first occurance).

Ordinarily when a function is called with arguements, the arguements are first evaluated and then the function is given their values (commonly referred to as *pass-by-value*). That is *not* the case here. Instead, the code itself is passed and the function decides when, if at all, to execute it (a much rarer form of arguement passing called *pass-by-name*). It may also execute the same code repeatedly, and because of side effects, that code could have a different value each time. For those familiar with LISP, this can be viewed as equivalent to each function taking (quote arg) instead of (arg) as the arguement, and then the function calling eval on the arguement. §2.3, which describes the functions, explicitely states when arguements are evaluated to eliminate ambiguity.

2.2 The Terminals

The GP contains three terminals:

- x - the integer to be factored

- sv0 - a settable value whose initial value is 0

- sv1 - a settable value whose initial value is 1

2.3 The Functions

The GP contains 8 functions:

- `set-sv0 (arg1)` - Evaluates `arg1` and sets `sv0` to its value and returns that value.

- `set-sv1 (arg1)` - Evaluates `arg1` and sets `sv1` to its value and returns that value.

- `mod (arg1, arg2)` (represented by `%`) - Evaluates `arg1` and then evaluates `arg2`. If `arg2` equals zero, it returns the value of `arg1` (This clause is different than the standard mod function, which is undefined when textttarg2 equals zero). Otherwise, it returns (value of `arg1`) mod (value of `arg2`).

- `add (arg1, arg2)` (represented by `+`) - Evaluates `arg1` and then evaluates `arg2` and returns the sum of their values.

- `equals (arg1, args2)` (represented by `==`) - Evaluates `arg1` and then evaluates `arg2` and returns one if their values are equal and returns zero otherwise.

- `less-yhan (arg1, arg2)` (represented by `<`) - Evaluates `arg1` and then evaluates `arg2` and returns onr if the value of `arg1` is less than the value of `arg2` and returns zero otherwise.

- `if-then-else (if_arg, then_arg, else_arg)` - Evaluates `if_arg` and if its value is greater than or equal to one it evaluates `then_arg` and returns that value. otherwise, it evaluates `else_arg` and returns that value.

- `do-while (arg_cond, arg_body)` - In `do-while` it is important to note that the `arg_body` always gets evaluated at least once, regardless of the value of `arg_cond`. (*) First `arg_body` is evaluated. Then `arg_cond` is evaluated. if the value of `arg_cond` is greater than or equal to one, it starts again from (*). Otherwise, the value of the most recent evaluation of `arg_cond` is returned. Additionally, so as to avoid infinite loops, each `do-while` is allowed a maximum of 100 iterations, and all `do-while` loops are collectively allotted a maximum of 200 iterations, after which the most recent value from evaluating `arg_body` is returned.

2.4 Sample Programs

In the original encoding of the problem, I had intended for there to be a `Seq` function which took two arguements, evaluated the first one and then evaluated the second and returned the result of the second. I had believed that this was necessary to solve the problem so that the settable variables could be initialized by the program. However, due to a coding error, this function was left out of the function set. The GP however was more creative than the author and managed to still find a solution. Therefor, instead of being human-produced, the sample program below was actually GP-produced and then human-optimized:

```
(do-while (% x sv1) (set-sv1 (+ sv1 (< sv1 x))))
```

In words, this program first increments `sv1` (whose initial value is 1) by 1 (the value of `(< sv1 x)`), unless `x = 1` (in which case `(< sv1 x)` has a value of 0), in which case `sv1` is not incremented. It then loops, where at each iteration it checks if `sv1` is a factor of `x` and if so, `sv1` is returned. If not, `sv1` is incremented and another iteration occurs.

Another possible, but incorrect, program is:

```
(+ (== x x) (== x x))
```

which always returns 2. This individual has a fitness of 50 because it gets all of the even integers correct, but none of the odd ones.

2.5 Evaluation of Fitness

Clearly the most natural measure of fitness is how often the program produces a correct factor. However, this alone was not sufficient and fitness had to encorporate some other aspects to nudge progress in the correct direction and to then reduce the size of the program to something managable and human verifiable. Each program starts off with initial fitness of $100 - number\ of\ hits$, and is then adjusted.
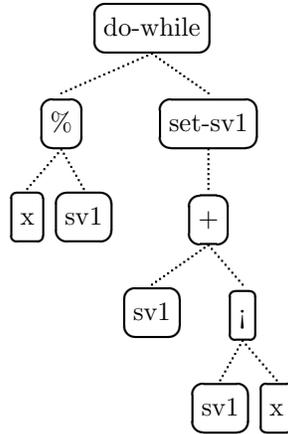
Figure 1: Sample parse tree of GP program `(do-while (% x sv1) (set-sv1 (+ sv1 (< sv1 x))))`
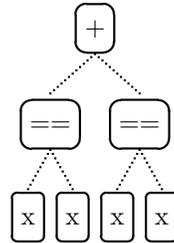

Figure 2: Sample parse tree of GP program `(+ (== x x) (== x x))`

2.5.1 Fitness cases

A general solution, for all positive integers, was found using only the integers 1 to 100 as fitness cases. This came as a great surprise to the author. The original intent had been to see how the GP did with numbers under 100, and then work up, but instead a general solution was found. The programs were evaluated on all 100 numbers, not a random sampling. Early on in the experiment, when the `do-while` function contained a serious bug, a solution which was only good for integers less than or equal to 100 was evolved. This solution was highly efficient and is discussed in section §5.1.

2.5.2 Forcing Use of `x`

One initial problem was that, because half of the fitness cases had 2 as a possible factor, the GP would persistantly find and return programs which reduced to 2 regardless of input and little progress was made. So a clause was added to the fitness measure which required that `x` appear in the program. This requirement does not force the program to actually use the `x`, since programs like `(== x x)` will be accepted despite returning 1 regardless of the value of `x`, but it does make it far more likely. If `x` did not appear in the program then that program was given the worst possible fitness.

2.5.3 Forcing Reduction of Program Size

When program size (defined by the number of nodes in the program's parse tree) was not kept in check several problems resulted. The first was speed - larger programs caused a major slowdown. Secondly, it seemed that it would be harder for a human to verify the correctness of a large, unwieldy program. Lastly, it was the author's intuition that a general solution should be short, while a long solution was likely to just contain lots of clauses for all possible inputs and not actually be a general solution. Programs were

penalized according to size, but the penalizations were always small enough that regardless of size, more accurate programs had higher fitness than less accurate programs.

Before a program acheives 100% accuracy the penalizations are less fine tuned. The goal was to favor smaller programs, while giving the GP as little opportunity as possible to 'game the scoring metric' by trying to optimize size over accuracy. Once a program achieves 100% accuracy the goal of the GP shifts to optimizing for size. Because hits now equals 100, the base fitness is 0, so the only thing adversely affecting a programs fitness is its size.

size	penalty
$size > 1000$	0.99
$1000 \geq size > 900$	0.90
$900 \geq size > 800$	0.80
$800 \geq size > 700$	0.70
$700 \geq size > 600$	0.60
$600 \geq size > 500$	0.50
$500 \geq size > 400$	0.40
$400 \geq size > 300$	0.30
$300 \geq size > 200$	0.20
$200 \geq size > 175$	0.175
$175 \geq size > 150$	0.150
$150 \geq size > 125$	0.125
$100 \geq size > 100$	0.10

Table 2: Size penalties for programs with less than 100% accuracy.

3 Parameters

The final run used a population of 5,000. This population size was selected because it was large enough to offer breeding diversity, but still small enough that the program could run in a reasonable amount of time. The maximum number of generations was set arbitrarily large (10,000) with the intent that the GP would terminate because an optimal solution had been found, and not due to too many generations.

Elitism was used, and the top 10 individuals were kept each round. Part of the motivation for thise was that the elitism, in conjunction with the size penalties of section 2.5.3, would promote smaller program sizes.

The ECJ defaults for crossover, reproduction and mutation were used. There is a 90% probability of crossover. The crossover style is simple subtree crossover. Individuals for crossover are selected using tournament selection with a tournament size of seven. There is a 10% probability of reproduction, where the individual being reproduced is copied to the next generation unchanged. Similarly to crossover, individuals are selected for reproduction using tournaments of size seven. There is no mutation. Other probabilities for crossover and mutation were tried, but the defaults worked sufficiently and converged at a reasonable speed, so they were kept for the final run.

4 Results and Discussion

All of these experiments were run on a single-processor Pentium 4 2.4GHz Linux workstation with 512MB RAM. Sean Luke's Evolutionary Computation and Genetic Programming System is written in Java, as was all of the author's code. It was compiled and run using Java 1.4.2_02. The final, successful, run took approximately one hour.

The final, optimal individual found was:

```
(do-while sv0 (do-while (% x (if-then-else x sv1 (set-sv0 sv1))) (set-sv1 (+ (+ sv1 sv0)
                              (< sv1 x)))))
```

size	penalty	size	penalty
$size > 1000$	0.99	$240 \geq size > 230$	0.23
$1000 \geq size > 900$	0.90	$230 \geq size > 220$	0.22
$900 \geq size > 800$	0.80	$220 \geq size > 210$	0.21
$800 \geq size > 700$	0.70	$210 \geq size > 200$	0.20
$700 \geq size > 600$	0.60	$200 \geq size > 190$	0.19
$600 \geq size > 500$	0.50	$190 \geq size > 180$	0.18
$500 \geq size > 400$	0.40	$180 \geq size > 170$	0.17
$400 \geq size > 390$	0.39	$170 \geq size > 160$	0.16
$390 \geq size > 380$	0.38	$160 \geq size > 150$	0.15
$380 \geq size > 370$	0.37	$150 \geq size > 140$	0.14
$370 \geq size > 360$	0.36	$140 \geq size > 130$	0.13
$360 \geq size > 350$	0.35	$130 \geq size > 120$	0.12
$350 \geq size > 340$	0.34	$120 \geq size > 110$	0.11
$340 \geq size > 330$	0.33	$110 \geq size > 100$	0.10
$330 \geq size > 320$	0.32	$100 \geq size > 90$	0.09
$320 \geq size > 310$	0.31	$90 \geq size > 80$	0.08
$310 \geq size > 300$	0.30	$80 \geq size > 70$	0.07
$300 \geq size > 290$	0.29	$70 \geq size > 60$	0.06
$290 \geq size > 280$	0.28	$60 \geq size > 50$	0.05
$280 \geq size > 270$	0.27	$50 \geq size > 40$	0.04
$270 \geq size > 260$	0.26	$40 \geq size > 30$	0.03
$260 \geq size > 250$	0.25	$30 \geq size > 20$	0.02
$250 \geq size > 240$	0.24	$20 \geq size > 10$	0.01

Table 3: Size penalties for programs with 100% accuracy.

However, this result has three easy optimizations which do not change the underlying algorithm, but instead make it clearer for the reader and easier to clearly discuss. First of all, in the code snippet:

```
(if-then-else x sv1 (set-sv0 sv1))
```

We know that x is always positive (the fitness cases, 1 - 100 are all positive integers), so the then clause is always evaluated, so this can be reduced to

```
sv1
```

After the previous reduction is made, it becomes clear that set-sv0 is never called, so sv0 always retains its initial value of 0. Therefor, code snippet:

```
(+ sv1 sv0)
```

is always equal to

```
sv1
```

Additionally, the outer do-while loop will always execute its body (the inner do-while loop) once and then evaluate sv0 to 0, which will cause it to return the result from executing the body after only the first iteration. This program is therefor equivilent to the programming containing only the inner do-while loop:

```
(do-while (% x sv1) (set-sv1 (+ (+ sv1 sv0) (< sv1 x))))
```
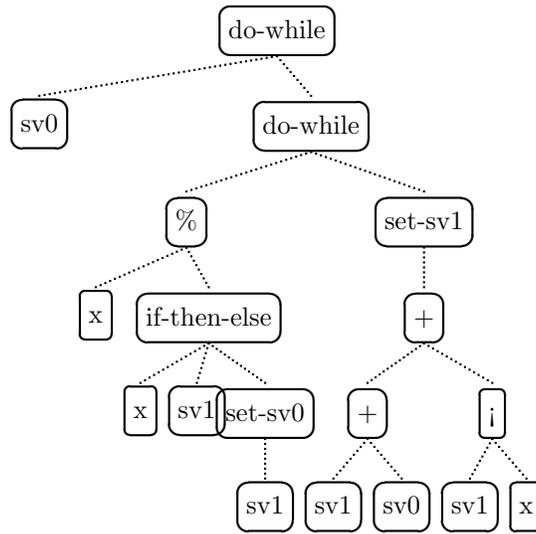
Figure 3: Sample parse tree of solution found by GP (`do-while sv0 (do-while (% x (if-then-else x sv1 (set-sv0 sv1))) (set-sv1 (+ (+ sv1 sv0) (< sv1 x)))))`)
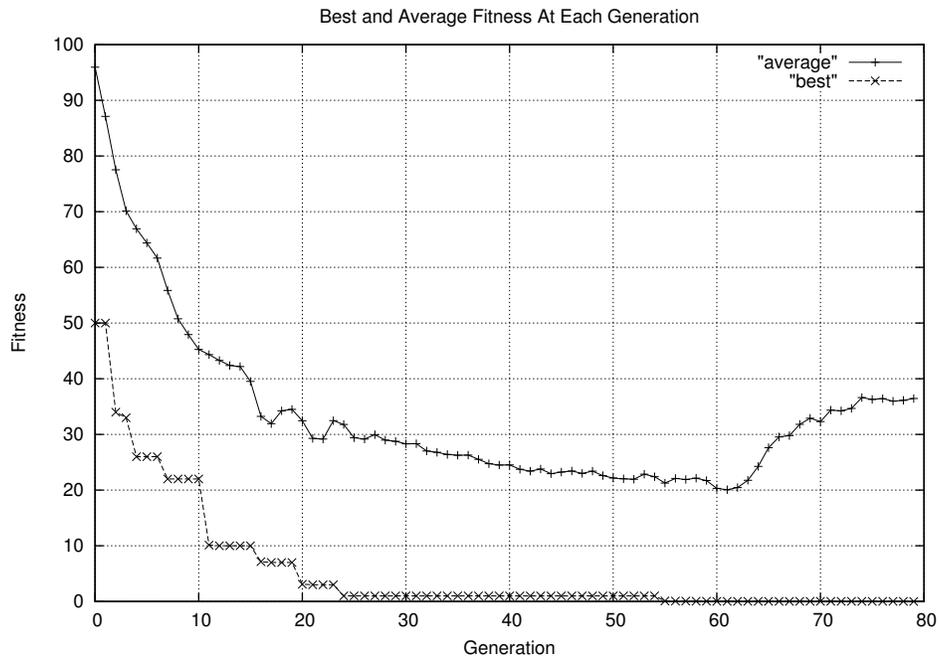

Figure 4: Best and average fitness over all generations

4.1   Statisitics

The GP took 55 iterations to produce an individual which got all 100 hits and then another 24 iterations to get one of accceptable size that was verifiably generally correct. Figure 4 shows the evolution of best and average raw fitness for the entire run.

   It is interesting to note that once the GP started optimizing for size instead of accuracy, the average fitness got increasingly worse.

4.2   The Evolution of the Search

The best individual of generation 0 got 50 hits:

```
(set-sv1 (+ (if-then-else (== (if-then-else sv1 sv0 sv0) (+ sv1 sv1)) (< (set-sv0 x) (<
sv0 sv0)) (% (set-sv0 sv1) (do-while sv0 x))) (< (set-sv1 (== x sv0)) (+ (+ sv0 x) (+ sv0
                                  sv0)))))
```

This individual, in a very convoluted way, always returns 2 and gets the even-parity fitness cases correct. By generation 2 the GP is making use of parity and getting multiples of 2 and 3 correct:

```
(+ (set-sv0 (+ (if-then-else x sv1 x) (if-then-else x sv1 (set-sv0 x)))) (do-while (% (%
(do-while sv1 sv0) (% sv1 sv1)) (do-while (set-sv1 sv0) (set-sv0 x))) (% (% (% x sv0) (==
                              x sv1)) (set-sv0 (< x x)))))
```

At generation 55 we get the first 100% accurate indivual:

```
(do-while (== (if-then-else (< sv1 x) (+ sv1 (< sv0 sv1)) (if-then-else sv0 (set-sv1 (%
x (if-then-else x sv1 (set-sv0 (% x sv0))))) x)) (< sv1 x)) (do-while (% x (if-then-else
 x sv1 (set-sv0 x))) (set-sv1 (if-then-else (if-then-else x sv1 (% x (if-then-else x sv1
x))) (+ (set-sv0 (+ sv1 (< sv1 x))) (% (% (% x (set-sv0 x)) (+ x sv0)) (% x sv1))) (+ sv0
 (+ (do-while (% x (if-then-else x sv1 (% x sv0))) (set-sv1 (if-then-else (if-then-else x
                sv1 (set-sv0 sv1)) (+ sv1 (< sv0 sv1)) x))) sv1))))))
```

And finially, at generation 79, we get the individual discussed earlier at the begining of the section.

4.3   Remarks

The evolved solution is remarkably concise. The three inefficiencies discussed at the begining of the section would likely have evolved out if the GP had been allowed to continue running, because of the size-optimizing. However, this algorithm is not efficient. For instance, it tries all integers less than the number being factored instead of just the odd numbers (actually, odd numbers and 2). Also, it tries numbers which are clearly too large, such as numbers greater than half of the number being factored. However, this is not surprising because the GP optimized the solution for space instead of speed.

5   Failures and False Starts

5.1   Settable Variables and Maximum Iterations

One problem I encountered which stumped me for a long time was a bug with settable variables. In the problem specification I had declared them to have particular initial values, however, after each test on each individual I was not resetting the variables. This resulted in information being passed from run to run, and one particularly puzzling bug where programs which made no reference to x would have different return values when run on different values of x and even on the same value of x. A similar problem I had was forgetting to reset the number of iterations performed by `do-until`s in a program at the beginning of each run. The result was that almost immediately all of the allowed iterations were used up and no more `do-until`s would loop for any program under any circumstances. Remarkably, when using the fitness cases 1 through 100 the GP still managed to find a solution that was 100% accurate on the fitness cases (though not generally accurate), and remarkably efficient:

```
         (if-then-else (% x (set-sv1 (+ (+ sv1 (== x x)) (% (seq sv1 x) (+ sv1 sv1)))))
   (if-then-else (% x (set-sv1 (+ (+ (== x x) (set-sv1 sv1)) (== x x)))) (if-then-else (%
        (set-sv0 x) (set-sv1 (+ (+ sv1 (== x x)) (== x x)))) (set-sv0 x) sv1) sv1) sv1)
```

Note that the program creates its own constants with the use of `(== x x)`. Effectively, what the program does, is it mods the input by each prime number whose square is less or equal to than 100 (two, three, five and seven) because each composite number less than or equal to 100 must have one of these primes as a factor. If the result is zero then that prime number is returned. Otherwise the input must be prime, so it is returned.

5.2 Local Optima and Overfitting

Two other problems encountered, whose solutions is discussed in §2.5.2 and §2.5.3 are converging on local optima and overfitting on the fitness cases. The optima converged on was always returning two which was correct half of the time. This problem was solved by severely penalizing programs which did not refer to x. The second problem, overfitting, was what I feared would happen if the programs grew unboundedly and just had cases for each input. To solve that problem, programs where also penalized for being too large.

6 Conclusions

This paper has demonstrated how genetic programming can be used to find an algorithm for factoring positive integers. The algorithm found is respectable considering the objective of the project and the fitness measurement used. It was impressive that the GP managed to find a general solution to the problem with such a limited set of fitness cases. We also saw the GP perform admirably when given function sets less diverse than intended.

7 Future Work

I believe there are two main routes for further work. The first would be to evolve an algorithm which is a little smarter in how it looks for factors. Specifically, have the fitness measure penalize programs based on the number of iterations inside the `do-shile` loops so that faster programs are produced. A second area to look at would be attempting to use GP to evolve an actually efficient algorithm. A lot of people believe that such an algorithm may exist, but that no one has found it yet. Maybe a genetic program can find it for us.

8 Acknowledgements

9 References

Koza, John R. 1992. *Genetic Programming: on the programming of computers by means of natural selection* Cambridge, MA: The MIT Press 1992.