Automatic Generation of Prime Factorization Algorithms Using Genetic Programming

David Michael Chan
Department of Computer Science
Post Office Box 12587
Stanford, California 94309
dmchan@stanford.edu
(650) 497-6264

June 5, 2002

Abstract: This paper describes the application of the principles of genetic programming to the field of prime factorization. Any prime factorization algorithm is given one integer and must generate a complete list of primes such that, when multiplied together in varying degrees, produces the original integer. Constructing even a limited factoring algorithm in GP turns out to be extremely challenging and potentially impossible.

Introduction and Overview

Deriving an algorithm which scales better than *O(n)* to numerically factor integers into its constituent prime factors has been a crucial problem in recent times. Most encryption schemes, including RSA, depend on the fact that it is compute-intensive to factor large numbers in order to obscure sensitive data. Solutions to this problem include resorting to highly parallelized compute environments such as quantum computing, DNA-based computations, or massively-parallelized computing farms.

On the surface, attempting to derive an algorithm to factor integers into prime constituents seems to be a problem well suited for genetic programming. The goal is well defined: the genetic programming process must evolve an algorithm which takes an integer i and outputs n prime integer factors, such that when the $\prod$ product of these factors is calculated, it returns the original integer i. The fitness cases are similarly well defined: an individual is considered to be fit when its outputs are all prime integers that, when multiplied together, produces the original integer i. Traditional primality tests, such as the Rabin-Miller Strong Pseudoprime Test, are well understood and can be efficiently calculated for numbers up to 3.4×10^{14} (Rabin 1980). Additional fitness cases to evaluate compute-expense may be added in order to evolve higher performance algorithms; however, first, it is important to determine whether GP is capable of evolving a simple factoring algorithm.

Prime Factorization

The Fundamental Theorem of Arithmetic (unique factorization theorem) states that any positive integer can be represented in exactly one way as a product of primes. Mathematicians such as Euler and Fermat used to factor primes without computers at an astounding rate. Ever since these feats, there has been speculation that there was once a secret method of factoring primes that has since been lost.

Figure 1: Potentially Evolvable Direct Search Factorization Algorithm

```
(eval
  (for (sqrt x)
     (if (and (and (> i (/ x x)) (== (% i x) v0))
              (is_prime i))
       (set_v1 i)
       x
     )
  )
  v1
)
```

There are many algorithms to factor numbers into their primes, ranging in complexity and speed. The simplest algorithm is the *direct search factorization* algorithm. All possible factors (up to the floor of the square root of *n*) are tested to see if they divide *n* properly. Variations exist where multiples of smaller primes are excluded from the test list to eliminate obvious non-primes. Obviously, this method is only practical for relatively small numbers. It achieves approximately *O(n)* performance if the primality test is constant test or is dominated by *O(n)*. These algorithms range in sophistication and efficiency all the way up to the Pollard-Strassen method, which is regarded as the fastest-known, fully proven, deterministic algorithm. It has the complexity $O\{exp[c(\log n)^{1/3} (\log \log n)^{2/3}]\}$.

It is widely considered in the mathematics world that this problem is "computationally hard" and that building a general-purpose algorithm to solve it is extremely difficult. In fact, it is considered so hard, several incredibly innovative and spectacular methods have been developed throughout the 20th century in order to improve the performance of solving this problem. These range from the *Elliptic Curve Factorization Method*, which uses random points on elliptical curves to factor, to using various uses of *sieves*, which eliminate factoring possibilities, thus leaving a smaller number of elements to process.

Despite its difficulty, we will attempt to use genetic programming to construct an algorithm to factor integers into its constituent primes.

Methods

The runs reported in this paper were designed around the standard genetic programming paradigm as defined in Koza (1992). The problem was coded in Java™ using Sean Luke's ECJ 8 Java™-based Evolutionary Computation and Genetic Programming Research System. The runs were executed on several Sun™ Blade 2000 machines, operating on Dual 1GHz UltraSPARC™ III processors in the Stanford University computing environment.

Table 1: Table for the Prime Factorization Problem

Objective	Find a mixed-typed function that performs a simple direct factorization over a range of fitness cases.
Terminal Set	X, I, V0, V1, RandPrime
Function Set	/, %, Sqrt, SetV0, SetV1, Eval, For, IfGreaterThan, IfEquals, IfLessThan, IfPrime
Fitness cases:	25 numbers, which are the products of randomly selected primes from (1,100)
Raw Fitness:	100 multiplied by the number of hits, plus bonus points for an output greater than 1, different from the fitness test case, and prime.
Standardized Fitness:	10,000 divided by Raw Fitness. If no points are given, standardized

	fitness is 10,000. If all hits are made, the individual has a perfect fitness: 0.0.
Hits:	The number of fitness cases that the individual returns either prime factor of the case.
Parameters:	- Population size, M = 1024, - Maximum number of generations, G = 101, - 89% crossovers, 10% reproductions, and 1% mutations were used on each generation. - Tree builder is the Strongly-typed Probabilistic Tree Creation 2 (PTC2) was used (Luke 2000). - All other parameters are standard, per Koza (1992).
Success Predicate	When the number of hits for an individual equals the number of test cases.

Objective

Because of the difficulty of constructing a full-blown, general purpose factoring algorithm, for the purposes of this paper, we will be attempting to evolve an individual that is able to take in a number that is the product of two primes and will return either one of those primes. This avoids the complication of maintaining a unique array with values that may or may not correspond properly to an answer. Our simplified approach allows us to gain some knowledge about GP's limits and abilities.

Terminals and Functions

- *V0*, *V1*, *SetV0*, *SetV1* are terminals and function that act as variables and internal state for each individual. *SetV** takes in one argument, stores the value given into *V**, and returns that argument. *V** returns the value stored in its state. This set of functions allows the individual to save a value somewhere in its execution when the main return value might be changing—for example, during a *For* iteration.
- *RandPrime* is a random prime generator terminal.
- *Eval* is a function that takes in two arguments, evaluates both trees, but only returns the value of the latter. It is a way for *SetV** to be run without deleterious effects. These can be strung together to get a more linear style of branching.
- *For* takes in two arguments, the limit up to which it should iterate, and the branch of functions to execute each time through. For each iteration, *For* updates a terminal *I* with the current iteration number (*i*). Thus, like traditional programs, the value of the iterative element is available for use in the functional branch.
- *IfPrime* takes in three arguments: the number being tested, and two conditional branches. If the first argument is prime, then the second branch is evaluated, else, the third branch is evaluated. *IfPrime* uses a custom-written, speed-optimized port of the primality testing found in Java™'s BigInteger class. It uses both the Rabin-Miller and Lucas-Lehmer primality tests to a certainty (13) that guarantees accurate primality tests for numbers up to 100,000.

Fitness and Hits

Our fitness cases are designed in a slightly unorthodox way. During a setup phase, a "prime-pool" of 100 times our training set is randomly generated, full of primes from [2, 100]. For each evaluation, our fitness cases are filled up by randomly selecting two primes from the prime-pool,

multiplying them together, and inserting the product into the individual to run. We define a successful fitness case (or hit) as when the individual returns either one of the original primes.

Due to the nature of the problem itself, it turns out to be difficult to be creative about designing fitness rewards that provide a gradual ramp up to a perfect individual. Each successful fitness case will reward the individual with a certain raw fitness amount (100), but each unsuccessful fitness case has a chance to redeem itself: if the return value is greater than 1, if the return value isn't equal to the test case value, and if the return value is prime, the individual is granted another 20 points.

After the raw fitness is determined, we perform some simple normalizing functions to achieve the range [0, +∞], where 0 is the best and +∞ is the worst. From there, we use ECJ 8's built-in fitness manipulation functions to achieve the adjusted fitness, etc.

Population

The population (1024) and generation sizes (101) were picked through trial and error and were influenced by the default values in ECJ 8's package. Everything else was standard, per Koza (1992), except for Sean Luke's PTC2 tree building algorithm, which provided very quick, very interesting initial generations (RRR).

Termination Criteria

We end when the maximum number of generations is reached or we have found a perfect individual.

Figure 4: Best Individual of Generation 0 of Run 24

```
(if_== x (set_v0 i) (sqrt x) rand_prime)
```

Figure 4: Best Individual of Generation 22 of Run 24

```
(if_prime (/ x (if_prime (/ x (eval v0 rand_prime))
     (for (sqrt (/ i v1)) rand_prime) rand_prime))
     (for (for x (/ x rand_prime)) rand_prime)
     rand_prime)
```

Figure 4: Best Individual of Run 24

```
(if_prime (/ x (if_prime (/ x (if_prime (/
     x (if_prime (if_< x (/ x rand_prime) (/ i
     (/ x rand_prime)) (set_v0 (if_< x i v1 x)))
     v0 rand_prime)) (for (sqrt (if_prime v0 v0
     x)) rand_prime) rand_prime)) (for (sqrt
(if_prime
     (/ x rand_prime) (for (eval (set_v0 (if_<
     (for x (if_> (if_== v0 (sqrt rand_prime)
         x rand_prime) v0 (if_prime rand_prime (if_<
         x i v0 i) (if_prime v1 (sqrt (/ x
rand_prime))
         rand_prime)) v1)) (sqrt (sqrt (set_v1 (eval
     (if_prime (% rand_prime x) v0 rand_prime)
     rand_prime)))) i (if_> rand_prime (/ (set_v1
     (if_< v1 rand_prime (/ x rand_prime) (if_prime
         rand_prime rand_prime (if_prime rand_prime
         rand_prime x)))) x) rand_prime v1))) (/ i
     x)) rand_prime) rand_prime)) rand_prime)
     rand_prime)) (set_v1 i) rand_prime)
```

Results

The best individual of generation 0 had an adjusted fitness value of 0.0654 (on a scale of [0, 1], where 1 is perfect). This individual essentially returned a random prime, which worked 7 times out of 25: 28%

In generation 22, the best individual had an adjusted fitness value of 0.115. This individual was the first to score 13 hits, and it quickly progressed to the maximum of 17 from there. The best individual across the next 70 or so generations stagnated at about 17 hits.

Our best individual of run 24 had 17 hits and an adjusted fitness value of 0.1453.

With the current function set and terminal set, this was the best individual. With a success rate of 68%, this individual was fairly robust compared to previous attempts and given the strenous testing given to each individual.

Discussion of Results

The stagnation of the individuals in this run demonstrates the limited fitness gradation present, both inherent in the problem and in the system in general. One can see in Figure 1 that it is certainly possible to write the ideal individual out with less than 20 nodes, but after 101 generations, it was not found. The GP system managed to converge to a maximum fitness value quickly, only after 65 generations, however, did not experience any improvement over the next 126 generations.

Perhaps what's interesting, is not the exact results achieved on this last run, but the behavior that GP has towards previous approaches to this problem.

The first iteration of this project involved sending the individual through fitness cases where it was given non-primes and expected to fill an array (through a function called *Insert*) with the prime factors of each case, such that, when multiplied together, would return the original fitness case. This endeavor turned out to be difficult to control. Individuals would evolve to merely insert the test case itself into the first element of the array and quit. This, of course, would yield a significant fitness measurement due to the fact that $x * 1 = x$. In other cases, individuals would insert into the array unchecked, and after a few generations, Java™'s memory limits were tested. Checks and limits were placed in an increasing amount of places, until the complexity of the project grew disproportionately to the hits coming in: zero.

The lack of results merited a re-consideration of the focus of the project. We simplified the project heavily, eliminating the array altogether, and limiting the testing range to numbers that are the products of two primes. This allowed simplification of the return value, as only one was now needed. (The other could be found by dividing the test case by the return value.) The overall complexity of the project and evaluation times dropped dramatically. However, true results did not start to show up until a random prime generator terminal (*rand_prime*) was introduced into the system. Almost immediately, results started showing up, but it was stagnating at around 4 or 5 hits. Once we added a floored square root function, the hits shot up to 17.

The project also started out with taking advantage of ECJ 8's strongly-typed tree constraint system by bringing over logical concepts from such traditional programming environments such as C, C++, or Java™. We attempted to bring over concepts such as mixing Boolean and numerical operators such as *And, Or, Nor, Nand, GreaterThan, LessThan, Equals* into functions such as *If* and *While*.

Once the complexity of the typing was removed and Boolean statements were reduced to functions such as *IfGreaterThan* and *IfPrime*, the speed to convergence increased significantly.

Conclusion

Through innovative, but simple function sets, we were able to get significant progress towards evolving an algorithm to do limited factoring in the manner described in the paper. It is important, though, to note that factoring in this way is probably "GP-hard" since its complexity is similar to a needle-in-the-haystack problem.

Future Work

It is likely unwise that future work be done on this line in GP. Genetic Programming does not seem to be suited to this nontrivial problem as it reaches the limits of what Genetic Programming is capable of handling.

Acknowledgements

I would like to thank Sean Luke of George Mason University for his extensive assistance with his incredible ECJ 8 Java™-based framework. His time and commitment to supporting his incredibly flexible and powerful framework is commendable.