Evolving Sorting Networks Using a Genetic Algorithm

Austin Stone, as46569, austinstone@utexas.edu

May 12, 2015

1 Abstract

This paper describes an exploration of the potential of genetic algorithms to evolve sorting networks.

2 Fitness Functions

The fitness function I used to evaluate a sorting network was based on the number of swaps away from a correct sort of a test, and based on the length of the genome. I used:

$1.0/max(swaps\_from\_correct, 0.1) + (1.0/max(swaps\_from\_correct, 0.1)) * (5.0 * (min\_genome\_size)/(length\_of\_genome))$

Where swaps_from_correct is the number of swaps that was needed to correctly sort the input list after being processed by the sorting network (determined by running merge sort of the post processed list and counting the swaps), min_genome_size is the minimum possible "correct" genome (chosen to be n log(n) where n is the length of the list to be sorted.[1]) Intuitively, the reason I chose to use the number of swaps from correct as a fitness proxy is because it allows for a more incremental evolution than the number of entirely-correctly sorted lists. If I were to use the number of entirely correctly sorted lists as a proxy, a genome which is one base pair away from a perfect solution could perform poorly and be rewarded a very low fitness value, whereas a genome that is far from correct but works well just for the limited test cases could be awarded highly. Furthermore, this allows a smaller number of test cases, since test cases aren't just evaluated as a binary correct or incorrect, but on a sliding scale.

[1] Based on my reading, the optimal possible length is some constant factor times n log(n). See http://arxiv.org/abs/1403.2777 for details.

I weight the genome length reward (i.e., $(min\_genome\_size))/(length\_of\_genome))$ by the num correct reward (i.e., $1.0/max(swaps\_from\_correct, 0.1)$) so this way progressively bigger rewards are given for shorter genomes as the correctness progresses. Without weighting the genome length reward by the correctness reward, early on in the evolution, the genome length reward will be the main driving force in the evolution, and later on the correctness will be the driving force as the correctness reward becomes increasingly large relative to the length rewards. Both of these situations have bad performance. The multiplicative constant of 5.0 was chosen based on experimentation; it seemed to give better performance than without a multiplicative constant.

The fitness function for the test cases was

$$total\_number\_genomes\_attempted/total\_number\_swaps\_from\_correct$$

where $total\_number\_swaps\_from\_correct$ was the cumulative sum of the swap error for all genomes which attempted to sort this test. E.g., if 3 genomes tried to sort a particular test, and genome α was 5 swaps from correct, genome β was 0 swaps from correct (i.e., sorted it perfectly), and genome ζ was 3 swaps from correct, this value would be $5 + 0 + 3 = 8$. $total\_number\_genomes\_attempted$ was simply the cumulative number of genomes that attempted to solve a test; in the above example it would be 3. This fitness function encourages tests that are solvable by most of the population. However, to allow exploration of different tests after a test has "settled" and most of the population has evolved the ability to solve a particular test, a test is killed off when
$total\_number\_swaps\_from\_correct/total\_number\_genomes\_attempted$
is less than the list length, i.e. when the average genome is less than n swaps away from solving a list of size n. This was chosen somewhat arbitrarily; it is definitely possible that tweaking this value in one direction or the other could give better performance.

3 Mating

I implemented simple crossover with locational, fitness proportional mating. Given two parents, the cross over point is selected to be a random number in between the average of their two genome lengths. I make sure that this cross over point does not splice a "gene," i.e. that it is divisible by two. The child then receives parent one's genome up to the cross over point, and parent two's genome after the cross over point.

Each genome is assigned an index into a list. Parents are selected by selecting the first parent proportional to its fitness relative to the entire list, and the second parent within a random, normally distribution range in the list from the first parent. This range is sampled from a gaussian distribution with a mean of 10. After the range is decided, the second parent is selected from this

range within the first parent proportionally to its fitness relative to the other genomes within this range. The purpose of this locational selection is that it allows different regions in a large population to evolve different "strategies;" the idea is somewhat akin to speciation. It keeps different regions separated so one strategy or genome does not dominate the entire population.

For test genomes, I simply select proportional to the fitness and do not use locational breeding.

4 Mutations

I allow three types of mutation. Delete mutations, which remove a gene (again by gene I two base pairs which encode a comparison), add mutations (which take a comparison and randomly copy it somewhere into the list), and basic mutations (which change a single nucleotide/number in the genome). I allowed mutation rates to evolve over time. To do this, I defined a meta-mutation rate and meta-mutation amount for each of these three mutation rates. The meta-mutation rate is defined to be .2 and the meta-mutation amount is defined to be .01. This means that during roughly 1/5 breedings, the mutation rate of the child will change by 1 percent either up or down.

For test genomes, I only allow simple, single base mutations. These also can evolve with a meta-mutation rate of .04 and a meta-mutation amount of .01.

I initialize all genomes to a first order mutation rate (the "actual" mutation-rate, not meta-mutation rate) of a randomly selected value between 0.0 and 0.05.

5 Results

The following two graphs were from a trial with a population size of 1000, a test population size of 20, 160 generations, and a list length of 16. To test that the genomes being evolved were correct, I wrote two functions in the "TestSortingNetwork.py" file. The function testSortingGenome runs through all permutations of possible inputs (actually, it relies on the Zero-One principle, see the wikipedia page http://en.wikipedia.org/wiki/Sorting_network) and returns True if the genome is correct for all permutations and false otherwise. The function testAllGenomes takes the output from my genetic algorithm and tests the best organism at each generation for correctness. While not all the best organism genomes are correct (since I only test on a subset of all possible inputs, this makes sense), I usually find several totally correct organisms. In the trial graphed below, I found 5 totally correct organisms. An example of one such genome is the following:

[0, 13, 13, 9, 1, 13, 7, 10, 9, 3, 1, 15, 5, 7, 2, 9, 10, 14, 3, 4, 5, 15, 14, 5, 13, 1, 3, 3, 2, 13, 10, 15, 11, 7, 2, 13, 7, 6, 6, 15, 6, 2, 3, 10, 8, 9, 1, 8, 8, 6, 4, 9, 15,

12, 8, 10, 10, 1, 0, 4, 1, 14, 7, 5, 8, 14, 3, 12, 9, 13, 3, 12, 3, 7, 10, 1, 3, 0, 11, 14, 13, 7, 10, 8, 0, 10, 6, 14, 9, 1, 15, 13, 10, 0, 6, 11, 15, 3, 0, 6, 1, 6, 7, 11, 10, 11, 10, 10, 0, 4, 15, 8, 0, 12, 11, 13, 7, 14, 1, 9, 12, 0, 2, 4, 14, 11, 0, 7, 7, 12, 15, 13, 13, 5, 11, 3, 2, 1, 0, 8, 12, 1, 12, 2, 15, 9, 5, 3, 15, 6, 14, 10, 3, 9, 3, 2, 7, 0, 10, 8, 0, 7, 10, 10, 12, 5, 7, 15, 3, 4, 7, 14, 15, 14, 1, 11, 0, 3, 14, 3, 10, 12, 5, 8, 0, 8, 2, 12, 12, 1, 6, 3, 5, 14, 6, 11, 8, 5, 2, 1, 11, 13, 9, 9, 2, 14, 12, 8, 15, 12, 0, 9, 12, 5, 6, 4, 4, 13, 2, 10, 4, 14, 3, 13, 12, 11, 1, 14, 0, 15, 6, 2, 7, 4, 12, 15, 0, 6, 15, 15, 1, 0, 15, 6, 7, 15, 10, 13, 15, 12, 9, 5, 13, 5, 4, 9, 6, 10, 9, 14, 6, 11, 14, 5, 5, 0, 1, 0, 1, 3, 3, 14, 8, 10, 9, 14, 6, 3, 3, 13, 14, 5, 10, 9, 6, 10, 9, 14, 6, 13, 14, 5, 5, 0, 1, 0, 1, 3, 3, 5, 13, 5, 15, 0, 1, 5, 13, 5, 15, 3, 1, 8, 3, 7, 7, 14, 3, 14, 3, 15, 15, 15, 12, 10, 5, 15, 3, 12, 0, 2, 4, 11, 1, 12, 0, 3, 12, 4, 3, 11, 3, 6, 3, 14, 3, 12, 15, 3, 2, 9, 9, 5, 10, 12, 12, 10, 9, 9, 14, 5, 15, 10, 3, 5, 15, 10, 3, 3, 7, 10, 3, 5, 15, 10, 3, 3, 7, 10, 3, 3, 4, 8, 3, 14, 9, 5, 15, 8, 5, 15, 3, 11, 3, 12, 0, 1, 1, 6, 6, 2, 4, 4, 3, 10, 1, 10, 4, 14, 3, 5, 8, 11, 11, 13, 11, 12, 12, 4, 3, 12, 0, 11, 1, 11, 9, 11, 12, 5, 8, 9, 7, 5, 0, 13, 1, 13, 9, 6, 5, 10, 3, 13, 6, 10, 10, 11, 4, 12, 4, 11, 6, 5, 8, 9, 7, 5, 0, 13, 1, 13, 9, 6, 5, 10, 3, 13, 1, 12, 6, 7, 0, 10, 10, 11, 4, 12, 6, 12, 2, 5, 0, 13, 1, 13, 9, 6, 5, 11, 4, 12, 6, 12, 2, 5, 0, 7, 0, 10, 10, 11, 8, 5, 4, 12, 6, 12, 2, 5, 0, 13, 1, 13, 9, 6, 5, 10, 3, 13, 6, 10, 10, 11, 4, 12, 6, 12, 3, 3, 2, 12, 2, 8, 2, 15, 5, 9]

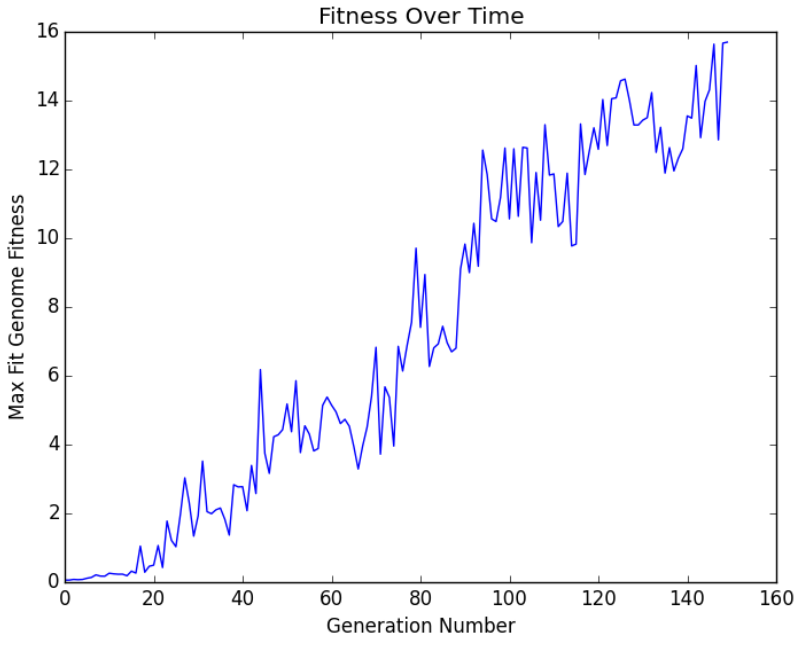

Figure 1: The above figure shows the max fitness per generation for a population of size 1000 with a sorting network size of 16. I ran this for 150 generations, and a population of 20 tests was used.

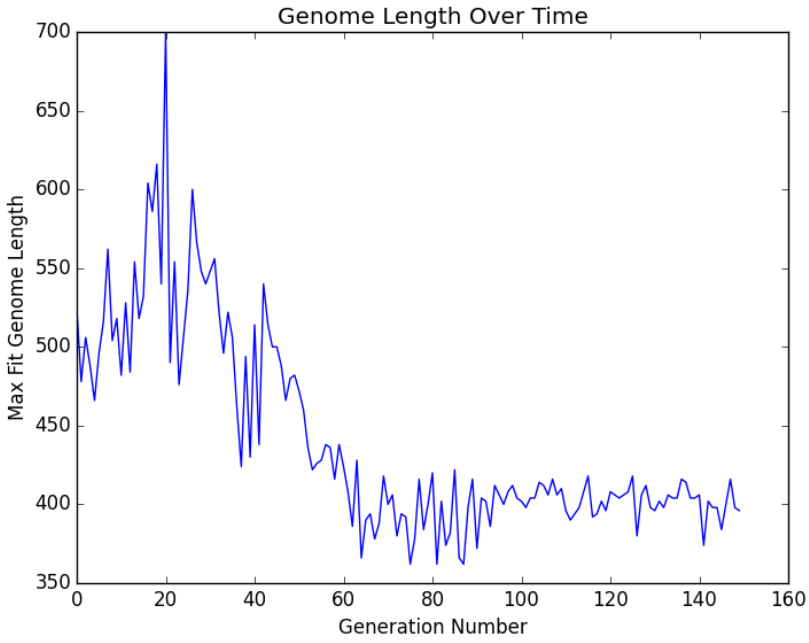

Figure 2: The above figure shows the max fitness genome length per generation for the same trial as displayed above.

6 Conclusions

Honestly, this seems like an incredibly deep area of exploration. In my own experiments, I was able to conclude that locational breeding seemed to perform better than simple fitness proportional breeding. I also observed that the "swaps from correct" metric seemed to work better than a binary "correct" or "incorrect" for evolving sorting networks of large lengths. My reasoning for this is explained above. I find the concepts at play here very fascinating; I wish I had more time to explore them. I spent a lot of time coding and testing ideas, so I didn't have much time to run a very large population for a very long time. I am curious how this would behave with a much larger population and given more time to evolve, especially since my graphs seemed to indicate that my population was still evolving at the end of the trial. Through biological evolution we have proof of what genetic algorithms can accomplish - it seems the concept has a lot of unrealized potential.