Genetic Algorithm: The Traveling Salesman Problem

Emily Strong

Final Draft: December 10, 2017

**Problem**

I created a genetic algorithm to find a solution to the Traveling Salesman Problem. The problem poses the situation: given a set of cities and distances between each of them, find the shortest path for a route that visits each city and returns to the originating city. Translating this to a graph model, it can be restated as given a complete weighted graph, find the Hamiltonian cycle with the smallest weight.

**Implementation Design**

*Genetic code:* Each base is an integer corresponding to a vertex. Each gene is an array of integers representing the order in which the vertexes are visited. While this may not be the most elegant way to represent the data, it simplifies mutation and crossing over and guarantees that every gene created will be a valid Hamiltonian cycle.

*Gene expression:* Each adjacent pair of bases corresponds to an edge, as well as the last and first bases. The edge weights are stored in a Bag data structure.

*Fitness function:* Since I am evaluating for the shortest path, the fitness function is the inverse of the net weight of the cycle. That is, f(g) = 1/w(g) where w(g) = Σ(edge weights). A cycle with a smaller weight will have a larger fitness value and vice versa.

*Mutation:* I am doing swap mutations of randomly selected bases. For example:

Parent: 1 | 7 | 2 | 4 | 5 | 3 | 6

swap indexes 1 and 2

Child: 7 | 1 | 2 | 4 | 5 | 3 | 6

I am also doing an insertion mutation, shifting three adjacent bases to a different location on the gene. This is done as:

Parent: 1 | 7 | 2 | 4 | 5 | 3 | 6

insert values starting at index 5 to index 2

Child: 1 | 5 | 3 | 6 | 7 | 2 | 4

Which version of mutation is performed on a child is selected at random.

*Crossing Over:* Since a pure crossing over would result in an invalid solution, I must effectively repair the gene afterward. To accomplish this, I randomly select pairs of genes, randomly select an index, and perform a swap mutation based on the value the other parent has at that index. For example:

Parent 1: 1 | 3 | 2 | 4 | 5 | 7 | 6

Parent 2: 2 | 7 | 1 | 3 | 6 | 5 | 4

swap index 3 ↑

In Parent 1 we find the index of the value that has index 3 in Parent 2 and swap that with the value at index 3. We repeat this process for Parent 2.

Child 1: 1 | 4 | 2 | 3 | 5 | 7 | 6

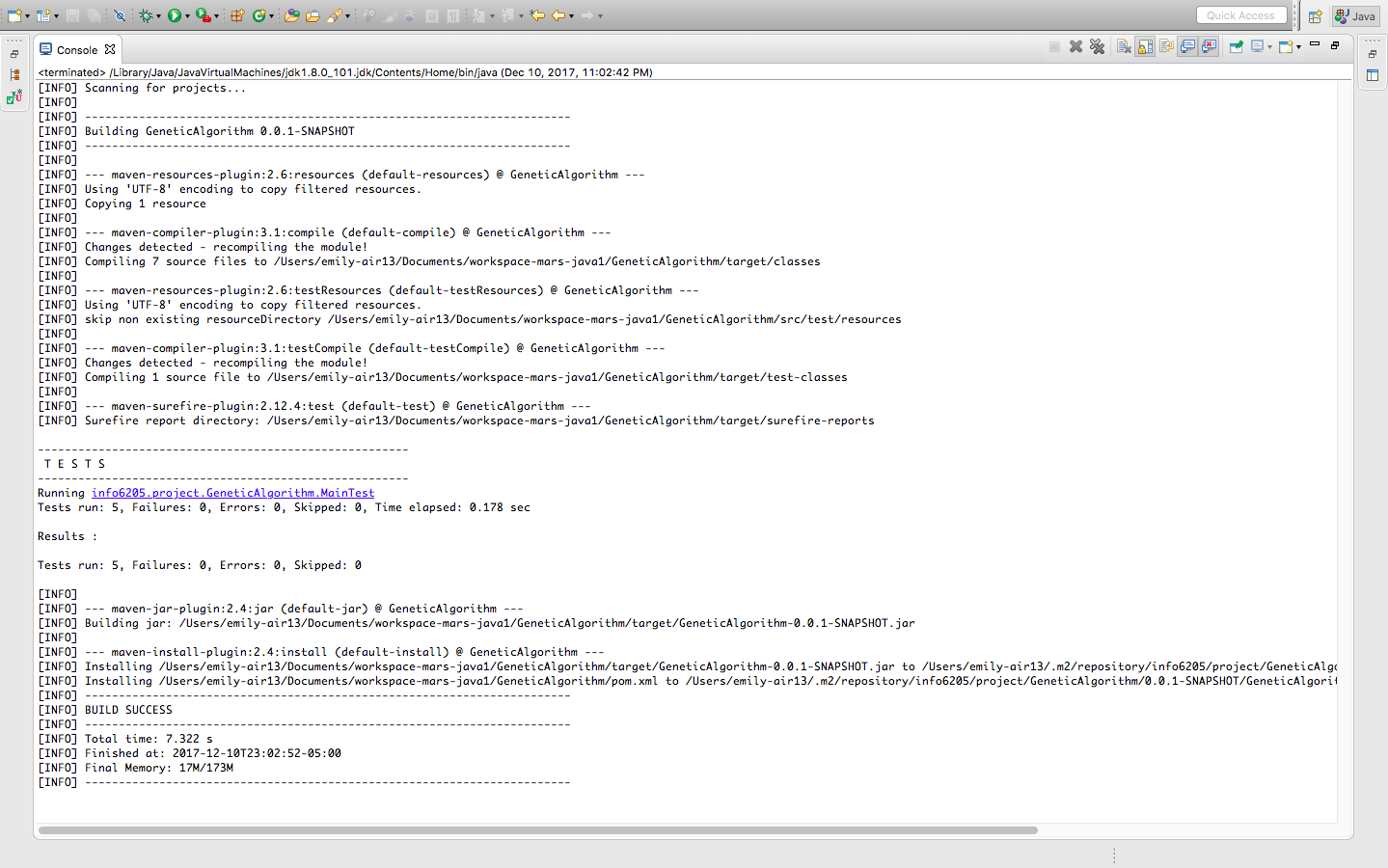
Child 2: 2 | 7 | 1 | 4 | 6 | 5 | 3

*Evolution:* The population is stored as a doubly linked list. The population is seeded with 4 individuals with genes that have been shuffled with the Fisher-Yates method. For each generation, all individuals sexually reproduce, doubling the population, and each child also experiences one of the two mutation methods. I then calculate the fitness function for each individual and write those values to an array. I perform an insertion sort on the array (selected since the graph sizes being evaluated are relatively small), randomly generate a culling ratio between 10% and 90% and remove that proportion of the weakest individuals from the population. The remaining individuals then reproduce, doubling the population again.

For each generation, after the culling is complete the number of survivors is logged along with the highest fitness score. The evolutionary process terminates after 100 generations or when the highest fitness score hasn't changed in 10 generations.

**Results**

My project builds and passes my unit test cases:

****

I ran the algorithm for graphs with 5 vertices, 25 vertices, and 50 vertices. The edge weights are randomly assigned values between 1 and 5, with a random seed so that each run will use the same graph and the results can be compared. The output from these runs is in log4j-application.log. Note: for v=25 there are a larger number of trials due to debugging. One trial also took such a long time to run I had to terminate it due to needing to complete the assignment on time. Since the smallest edge weight is 1, the maximum possible fitness score is 1/V. (Note: the first generation is generation 0, so the generations below are the generation id + 1)

|  |  |  |  |
| --- | --- | --- | --- |
| **Trial 1** | **5** | **25** | **50** |
| Generations | 19 | 49 | 55 |
| Final Score | 0.1667 | 0.0286 | 0.0125 |
| Time | 0s | 2s | 25s |
| Survivors | 28 | 689 | 7156 |
| **Trial 2** |  |  |  |
| Generations | 23 | terminated | 75 |
| Final Score | 0.1667 |  | 0. 0130 |
| Time | 0s |  | 5s |
| Survivors | 137 |  | 1988 |
| **Trial 3** |  |  |  |
| Generations | 21 | 14 | 100 |
| Final Score | 0.1667 | 0.0196 | 0.0143 |
| Time | 0s | 0s | 40 m 13 s |
| Survivors | 146 | 160 | 282042 |
| **Trial 4 - Corrected** |  |  |  |
| Generations | 11 | 23 | 34 |
| Final Score | 0.1 | 0.0196 | 0.0104 |
| Time | 1s | 0s | 5s |
| Survivors | 46 | 216 | 626 |

For the graph with 5 vertices, each trial produced the same cycle - 3, 1, 2, 0, 4. Since it is a small graph, this is most likely the actual solution to the problem for the graph generated. When rerunning the program with the corrected fitness function, the final fitness score changed but the same path was selected, consistent with this hypothesis.

One interesting effect is the size of the population on runtime and the solution. With a larger population the algorithm took longer to run, and even with culling the population size compounded, but with more individuals creating more opportunities for mutations and randomness, there was a positive relationship between population size, number of generations, and the final "most fit" score.

The relationship between number of survivors and run time appears to be logarithmic: