Genetic Algorithm: The Chinese Postman Problem- Route Inspection

Ashwath Shunti Mahabalagiri

Final Draft: April 15, 2018

**Problem**

We created a genetic algorithm to find a solution to the Chinese Postman Problem. The problem poses the situation: given a set of vertex and distances between each of them, find the shortest route for a postman who wishes to deliver letters along every road and come back to post office. Translating this to a graph model, it can be restated as given a connected undirected weighted graph, find the shortest route that visits every edge. This problem is proposed by Kwan Mei-ko in 1962, was called the Chinese postman problem (CPP).

**Solution Approach**

The route taken by the postman is considered as a connected undirected graph with edge-weights. **All the edges** should be included **at least once** in the tour. There exists a tour using each edge of the graph just once if the following property is satisfied: The degree of each vertex must be even. Thus, the method of solution for Chinese postman problem consists of finding a minimum cost augmentation of the graph that satisfies the sufficient property and then identifying the tour over the augmented graph.

The genetic algorithm is described as below:

1. Generate initial set of solutions (populations): P(0).
2. Calculate the fitness of all the individuals in P(0).
3. Generate offspring using cross breeding and mutations. Use only the most fit individuals in the parent population.
4. Fetch only the best of the population generated and destroy all the unfit offspring.
5. Repeat the steps 2 , 3 and 4 till there is no significant change in best fitness score.

**Implementation Design**

We know that the total number of the vertices which have odd degrees is even. Thus, we may use the code method as follows. 1) Divide total m odd vertices into m/2 pairs and add the shortest route between each pair to eliminate all odd vertices. 2) Adjust the scheme of the partnership to minimize the total cost of the augmentation of the graph. All odd vertices in the graph were picked out. Then we calculate the shortest route between each pair by Dijkstra's shortest path algorithm.

*Genetic code:* Each crossing is an integer corresponding to a vertex. Each gene is an odd vertex paired with the adjacent gene to add shortest route between each pair to eliminate all odd vertices. Each chromosome(solution) is an array of genes. The genes are stored in an array list data structure.

*Gene expression:* Each adjacent pair of crossing corresponds to an edge. The edge weights are stored in a Bag data structure. The sum of minimum distances between all the pairs will give the extra distance to be travelled in addition to all the edge-weights. So, different combinations of pairings (**different placements of genes inside the array list will give different additional distance** and hence, different gene-expression). **Less the extra-distance, better the solution**.

*Fitness function:* All odd vertices in the graph are picked out. Then we calculate the shortest route between each pair by Dijkstra's shortest path algorithm and put the result in the rank variable. A chromosome(solution) with a smallest weight(rank) is considered to be the fittest.

*Mutation:* The gene pairings are modified by pairing each gene with another gene which is at a particular distance. This distance is randomly chosen.

Mutant: 1 | 7 | 2 | 4 | 5 | 3 | 6 | 8

If the random index is 2,

Offspring: 1 | 4 | 7 | 5 | 2 | 3 | 4 | 6 | 5 | 8 | 3 |1 | 6 | 7 | 8 | 2 |

After repair: 1 | 4 | 7 | 5 | 2 | 3 | 6 | 8

*Crossing Over:* Random exchange of genes will result in odd node loss, or repetition of nodes which will result in an invalid route, hence invalid solution. A random cross over point is selected. All pair of genes are transferred from the first parent to offspring till this point from the beginning. And later the entire gene of the second parent is transferred to the offspring.

Parent 1: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

Parent 2: 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8

If the crossing point is 4,

Child1: 1 | 2 | 3 | 4 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 8

After repairing, child 1: 1 | 2 | 3 | 4 | 7 | 6 | 5 | 8

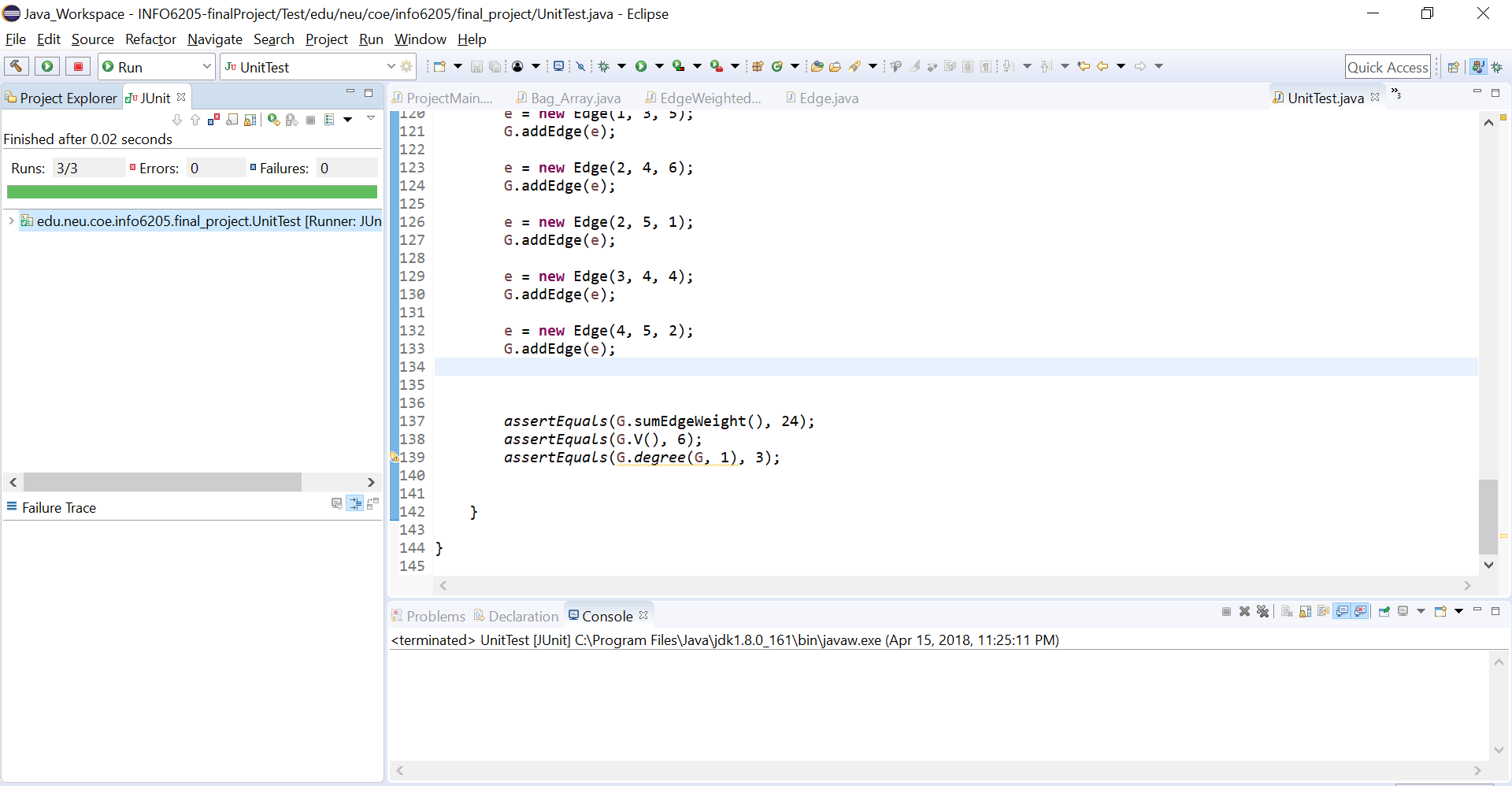
Similarly, the second child is created by copying the second last part of genes of second parent after the crossing point and appending all the genes of the first parent.

*Evolution:* The initial solution set (population) contains 6 individuals with genes that have been shuffled using system sort. For each generation, all individuals sexually reproduce – cross breeding, doubling the population. In addition to this, 2 of these offspring are subjected to mutation each generation. The rank of all the individuals is calculated after generating new population and the population is sorted in the increasing order of their ranks. The best of the population is kept for reproduction and the rest are culled. If there are more than 20 organisms, the best 20 will be survived and the rest are destroyed. If there are less than 20 individuals in a generation, the best individual and the individuals whose ranking is within the limit of 100 more than this rank score are kept for breeding. If there are no individuals other than the best for breeding, the second best is kept, irrespective of any constraints.

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 when the highest fitness score hasn't changed in 20 generations.

**Results**

**All the unit test cases are passed for individual methods:**



Below is the sample of the console output:



The program is executed for graphs with which are randomly generated with different vertex counts.

The edge weights are randomly chosen between 0 and 20.

(Note: the generated graph with 10 vertex had 4 odd vertices

graph with 10 vertex had 30 odd vertices and so on)

60 )

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Trial** | **10** | **60** | **110** | **160** |
| Generations | 22 | 35 | 42 | 63 |
| Shortest path | 109 | 1346 | 2471 | 4057 |

The above results are stable only to a certain extent because of the inefficiency to exhaust search all the combinations of solutions.

For node count more than 200, the program behaved unexpectedly, and terminated without coming to a solution.