**Traveling Salesman Project 4**

Tyler Hamilton

CECS, 1713904

Speed School of Engineering

University of Louisville

jthami05[@louisville.edu](mailto:NameID@louisville.edu)

1. **Introduction** (What did you do in this project and why?)

In this project, I implemented a genetic algorithm to solve the Traveling Salesman Problem. The purpose was to determine if a genetic algorithm would result in an efficient solution within a reasonable amount of time.

1. **Approach** (Describe algorithm you are using for this project)

The genetic algorithm I used is simple in concept, but multileveled and complex in practice. Initially, the list of cities is read in from a file the user is prompted for. The city list is placed into a list (TourManager), and the user is prompted for the number of cities the population will hold and the number of generations that will be evolved. A population of tours is generated randomly using this list of cities. Then, the process of evolution occurs for the given number of generations.

Evolution begins by taking the top 20% of the initial population and retaining them throughout the execution of the program. In my code, this is referred to as elitism and, at small expense to genetic variation, drastically increases the efficiency of the TSP solutions.

After this, parents are selected by being the fittest individuals in a randomly selected population, and the crossover function is applied to create children tours. The crossover algorithm takes a randomized substring of cities from the first parent, and then fills in the remaining slots with cities from the second parent.

The mutation algorithm is then applied. For my program, I found that an inversion mutation was more efficient than a swap mutation, and created more genetic variation. This inversion mutation selected a randomly sized substring from a child tour and inverted its position (so 1-2-3 would become 3-2-1). The new population would be returned and the evolution process would start over again.

Finally, the fittest member of the population is declared the winner and its distance is calculated.

No pruning of inefficient solutions occurred in this project, though it is noted that if it were, the execution of the program could be further optimized.

1. **Results** (How well did the algorithm perform?)

I ran 4 different scenarios with this algorithm. I modified the mutation process and the population/generation size. What I found was that the inversion mutation excelled compared to the swap mutation by far. However, even 2000 generations with a population of 70 tours left a few crossed paths in my tour. 3000 generations were enough to optimize the solution.

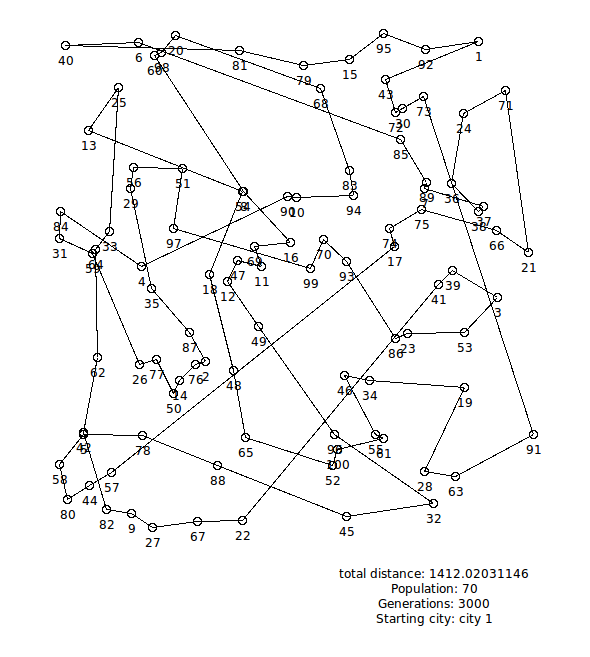
The algorithm performed fairly slowly. With 2000 generations, my code took almost 18 minutes to run, and with 3000 generations it took almost 35 minutes. The mutation used appears to have no noticeable difference on the running time of the process; it appears to be solely based on the population and number of generations. Increasing the population seems to drastically increasing the running time of the program.

* 1. **Data** (Describe the data you used.)

I used the .tsp file given in the project. The name of the file to be read was passed in by the user via prompt. I ignored the first sections of the files and read the remaining lines in, as IDs, x-values, and y-values for *city* objects.

* 1. **Results** (Numerical results and any figures or tables.)

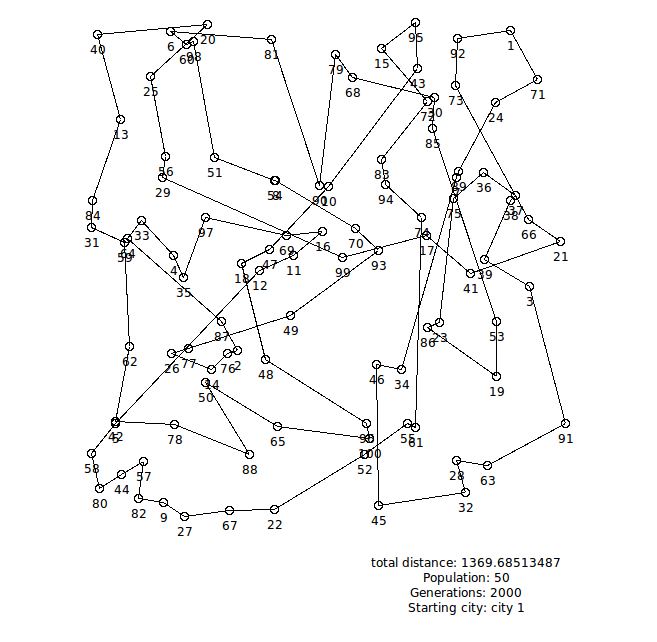
Population size 70, 3000 generations, swap mutation:



Distance: 1412.0203

Time taken: 2094.527071 seconds.

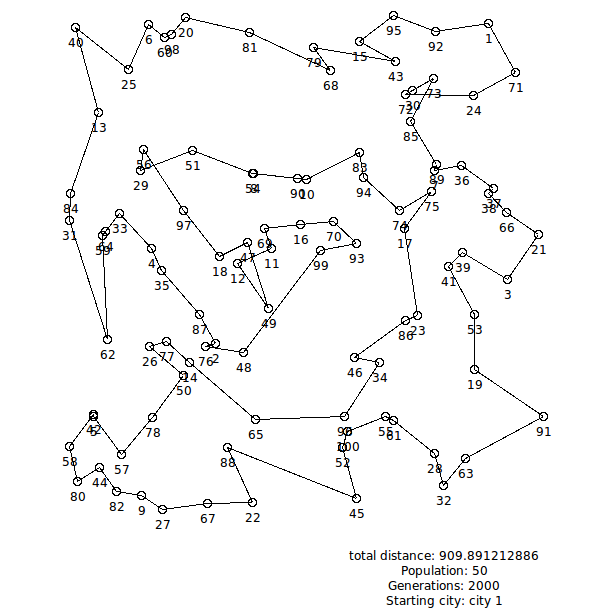
Population size 50, 2000 generations, swap mutation:



Distance: 1369.6851

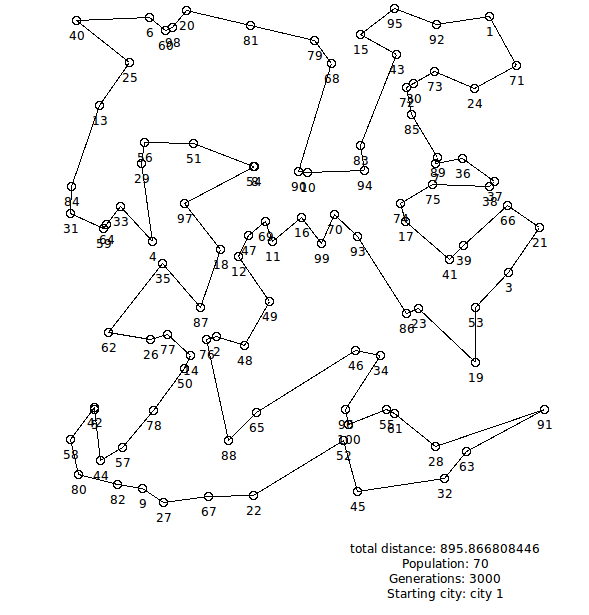
Time taken: 1086.110604 seconds

Population size 50, 2000 generations, inversion mutation:



Distance: 909.8912

Time taken: 1073.660155 seconds

 Population size 70, 3000 generations, inversion mutation:

Distance: 895.8668

Time taken: 2091.781388 seconds

1. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)

In my opinion, using a genetic algorithm as a problem solving technique will lead to efficient solutions to many complex problems; however, the largest limitation with this approach is runtime required for greater and greater amounts of data to work with. Excellent hardware is definitely a necessity to run the algorithm with large datasets.

I learned that genetic algorithms have potentially innumerable programming applications. If a problem can be formulated completely in code, with a mutation algorithm, crossover algorithm, and defined goals, it can be evolved.

The most difficult part of this project was without a doubt running an efficient solution without using several thousand generations more than I already use. For this aim I utilized a property in my algorithm called *elitism*, where I retained the fittest individuals of my previous generation into the next generation. This was instrumental in making my code fast enough to reasonably create an efficient solution.

The swap mutation I used swaps two cities selected at random in the tour. The inversion mutation and takes a randomized substring from the tour and flips it (1-2-3 → 3-2-1).

The improvement curves of my algorithm are as such:

Illustration 1: Population = 70, Generations = 3000, Inversion Mutation TSP Solution

Illustration 2: Population = 50, Generations = 2000, Inversion Mutation TSP Solution

Illustration 3: Population = 50, Generations = 2000, Swap Mutation TSP Solution

Illustration 4: Population = 70, Generations = 300, Swap Mutation TSP Solution

1. **References** (If you used any sources in addition to lectures please include them here.)

http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3