Traveling Salesman Problem (TSP) – Genetic Algorithm

Karthik Malyala
Computer Science & Engineering
Speed School of Engineering
University of Louisville, USA
ksmaly01@louisville.edu

1. Introduction

In this project, the Traveling Salesman Problem (TSP), a famous non-deterministic polynomial-complete (NP-Complete) problem, was explored using a Genetic Algorithm to find the shortest and most cost-efficient route for a salesman to be able to visit every single city from the starting city. This approach provided as supplement to the Closest Edge Insertion, Breadth-First Search (BFS) & Depth-First Search (DFS), and Brute Force approaches that were explored in the last three projects to analyze differences between the algorithm runtimes when dealing with complex TSP datasets where approaches like Brute Force, BFS, DFS, or Closest Edge Insertion would be inefficient as the number of cities increase, as seen in the last few projects. The above task was completed using Python 3.9 and its accompanying libraries in the PyCharms IDE using a Lenovo ThinkPad X390 Yoga with Core i7 processing and 16 GB RAM.

2. Approach

To implement the Genetic Algorithm, there were several fundamental concepts of evolution presented as functions that were combined to generate the optimal cost and route around the cities presented. Similar to how nature has the ability to learn and adapt without any instruction by finding good chromosomes blindly, the genetic algorithm also acts in the same manner. More specifically, this algorithm measures the fitness of individual chromosomes to carry out reproduction wherein a crossover function exchanges sections of two single chromosomes while a mutation function randomly swaps the gene value within a chromosome. With respect to the Traveling Salesman Problem:

- Gene - the set of coordinates for a city
- Individual Chromosome - a random route
- Fitness – the total distance of an individual chromosome (route) above
- Population – the set of all individual chromosomes (routes) above
- Parents – two random routes that are paired to reproduce a new route
- Mating Pool – the set of parents used to create the next generation (with elitism)
- Mutation – Function to cause variation in our population by swapping some cities in certain chromosomes(route)

For the purpose of this project, two different settings were observed for two different parameters. In this report, the two variable parameters explored were population size and mutation rate. I wanted to focus more on how population size and mutation rate would affect

each other on the output of the optimized cost rather than using two different crossover methods and analyzing their differences. The following datasets use these variable parameters as shown below:

1. Dataset 1A - Population Size of 50 and Mutation Rate of 0.0005
2. Dataset 1B – Population Size of 50 and Mutation Rate of 0.001
3. Dataset 2A – Population Size of 100 and Mutation Rate of 0.0005
4. Dataset 2B – Population Size of 100 and Mutation Rate of 0.001

To begin the genetic algorithm process, an initial population was created using random routes in a list of size of the given population size. After getting a population of various chromosomes, the fitness function will determine the total cost of each route (chromosome) in the population. Based on the results of the fitness function, the rankIndividuals function will then rank each route (chromosome) based on its fitness weight. Once the rankings of each chromosome have been computed, the selection function then does a selection by first retaining the best performing individuals from the current generation and then proceeds to select other individuals based on their fitness weights using the Roulette Wheel Selection method. Next, the matingPool function picks out the individuals selected by their fitness weights above and prepares a pool of parents for reproduction.

After the individuals have been filtered out by their performance, the crossover function then 'breeds' the next generation by swapping parts of the genes from each parent and produces a child. The breedPop function takes the matingPool computed above and 'breeds' a new population by calling the crossover function for the remainder of the mating pool after retaining the elites. Once the new population has been 'bred' through crossovers, mutation is performed in which a random pair of genes (cities) of an individual are swapped to diversify the solutions even more. This mutation process occurs on the basis of the given mutation rate. Mutation can be applied on any individual in the given population if the mutation rate allows to do so. Now that every function within the realm of evolution has been defined, the newGen function creates the next generation by calling the above functions again.

3. Results

The algorithm gave very near-to-optimal solutions, but none of them were optimal as seen below. The route graphs show several intersections (crossovers) which shows us how the path is not optimal enough and could be lower. This is due to the randomness of the algorithm which is discussed later on.

3.1 Data

The data being inputted was provided in the form of a .tsp file which had included a list of cities and their corresponding x and y coordinates. More specifically, there was one provided dataset with the coordinates of 100 different cities. The file was split into a specification part and a data part. While the specification parts contained descriptive information of the datasets, the data part contained all the vital information like Node Coordinates under NODE_COORD_SECTION, as seen below (100 Cities):

```
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
5 9.006012 81.185339
6 20.032350 2.761925
7 77.181310 31.922361
8 41.059603 32.578509
9 18.692587 97.015290
10 51.658681 33.808405
11 44.563128 47.541734
```

3.2 Results

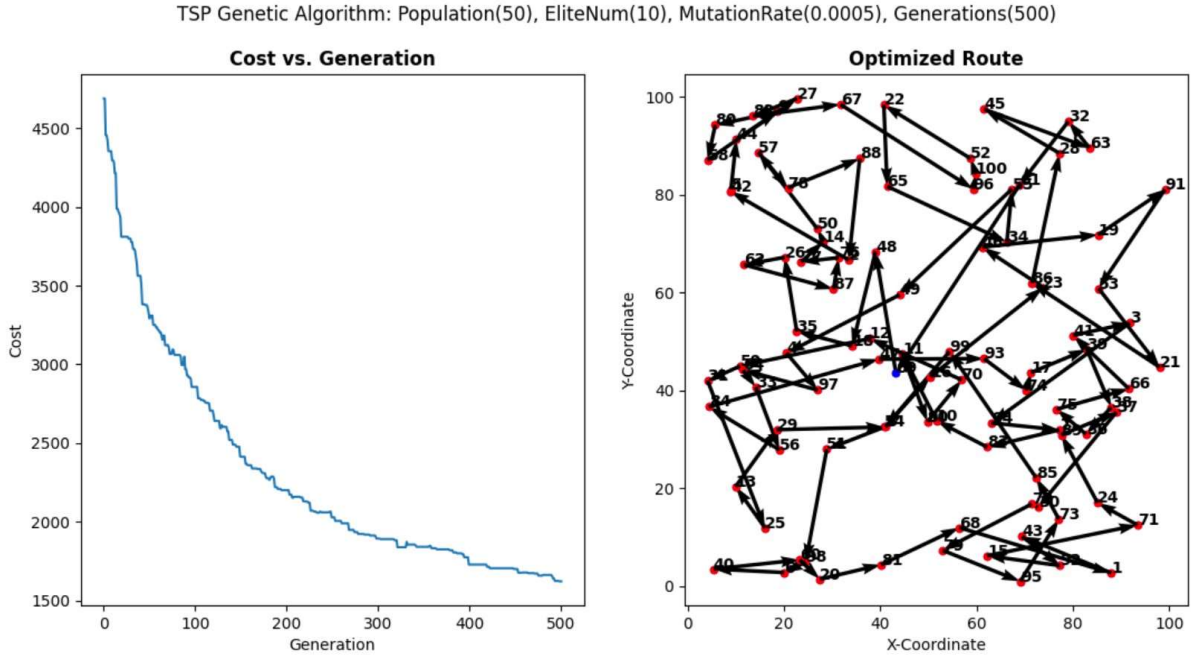
Dataset 1A - Pop(50) MRate(0.0005)	
Run 1	1123.02948
Run 2	1057.707117
Run 3	1204.152912
Run 4	1152.968153
Run 5	1098.369251
Min	1057.70712
Max	1204.15291
Average	1127.24538
Std. Deviation	49.52495165

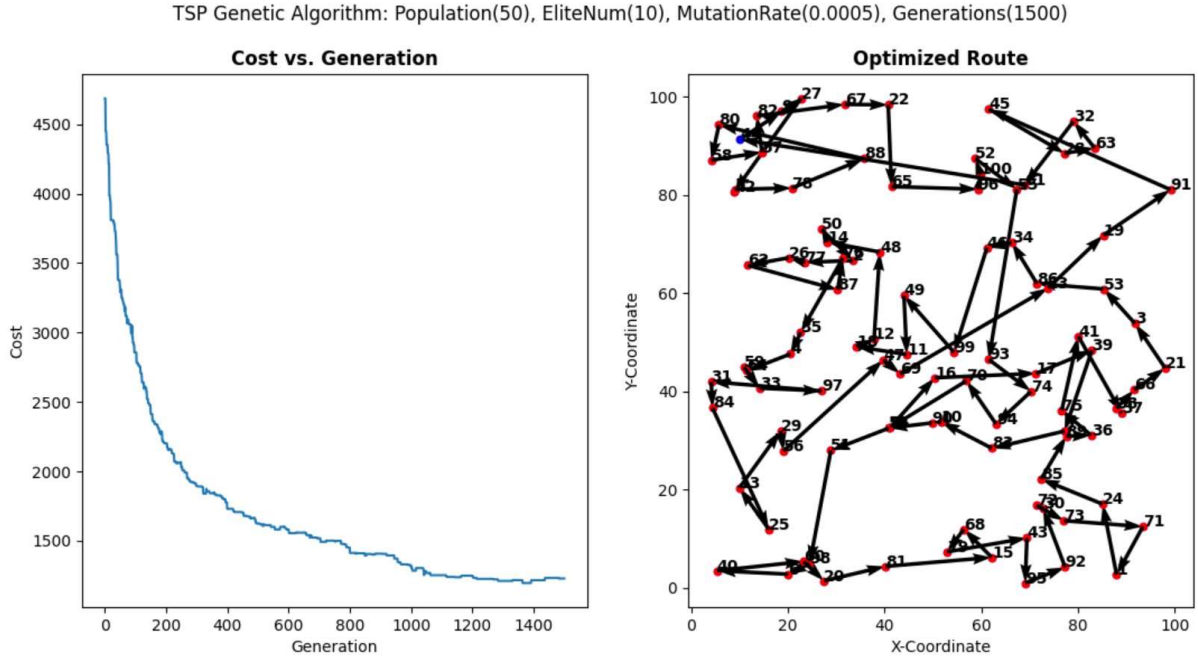
Dataset 1B - Pop(50) MRate(0.001)	
Run 1	1214.88481
Run 2	1112.732109
Run 3	1322.014298
Run 4	1094.137242
Run 5	1126.52489
Min	1094.13724
Max	1322.01430
Average	1174.05867
Std. Deviation	84.81212602

Dataset 2A - Pop(100) MRate(0.0005)	
Run 1	1121.76074
Run 2	1165.267542
Run 3	1041.840087
Run 4	1121.76074
Run 5	1097.695771
Min	1041.84009
Max	1165.26754
Average	1109.66498
Std. Deviation	40.31807969

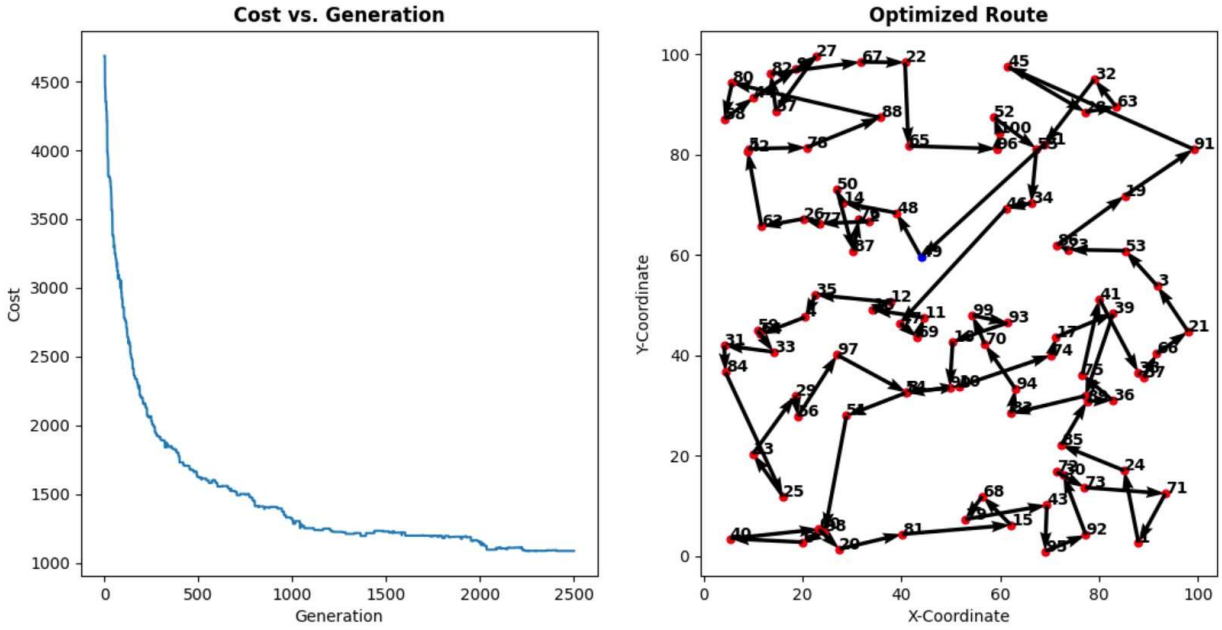
Dataset 2B - Pop(100) MRate(0.001)	
Run 1	1067.018418
Run 2	1101.720264
Run 3	1209.878424
Run 4	1078.323694
Run 5	1100.69122
Min	1067.01842
Max	1209.87842
Average	1111.52640
Std. Deviation	50.93144857

Dataset 1A

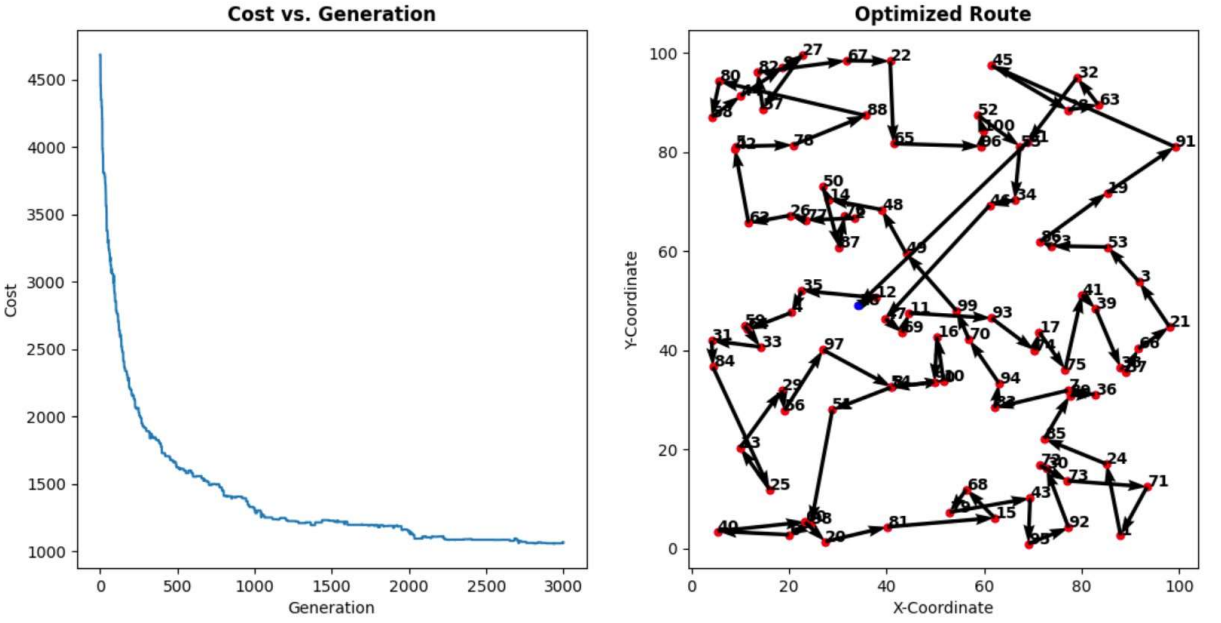

TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.0005), Generations(500)
Cost vs. Generation
Cost
Generation
Optimized Route
Y-Coordinate
X-Coordinate


TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.0005), Generations(1500)
Cost vs. Generation
Cost
Generation
Optimized Route
Y-Coordinate
X-Coordinate

TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.0005), Generations(2500)



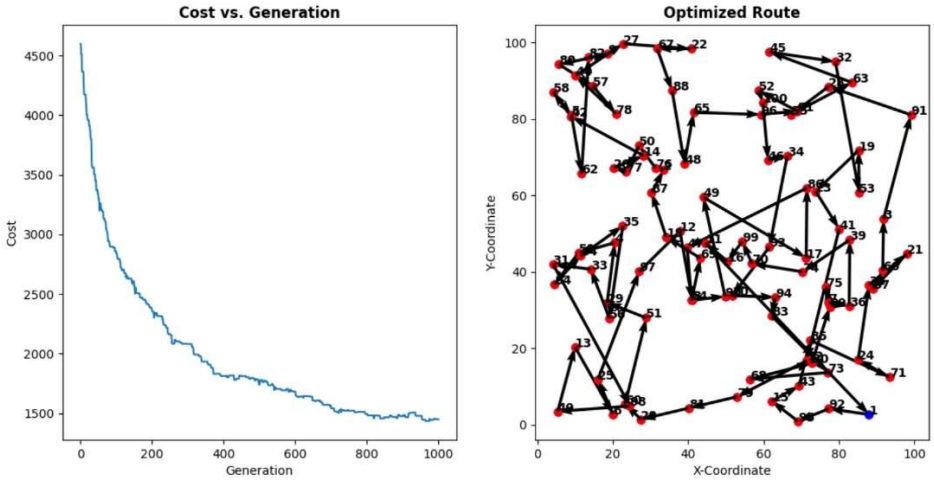
TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.0005), Generations(3000)



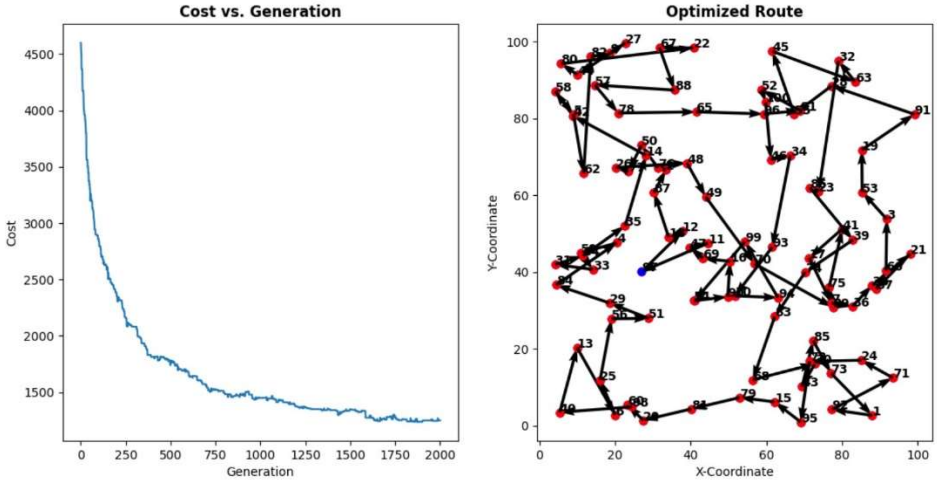

```
Final Dist: 1057.7071172666729
```

Dataset 1B

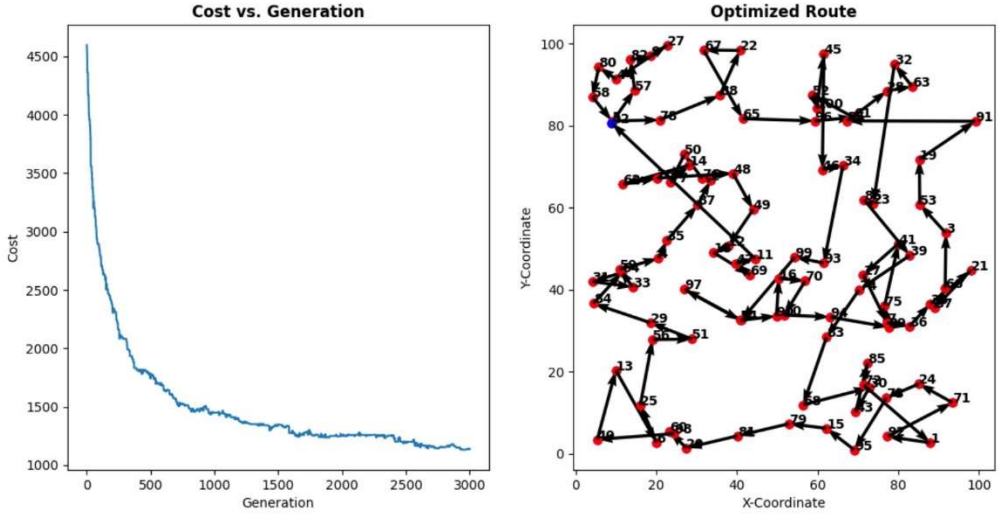
TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.001), Generations(1000)



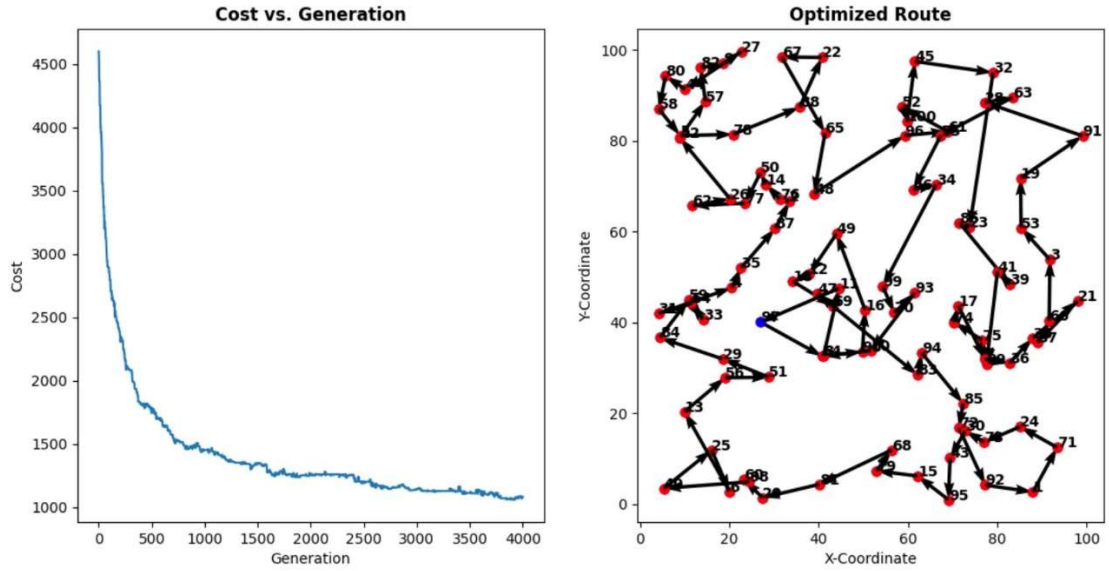
TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.001), Generations(2000)



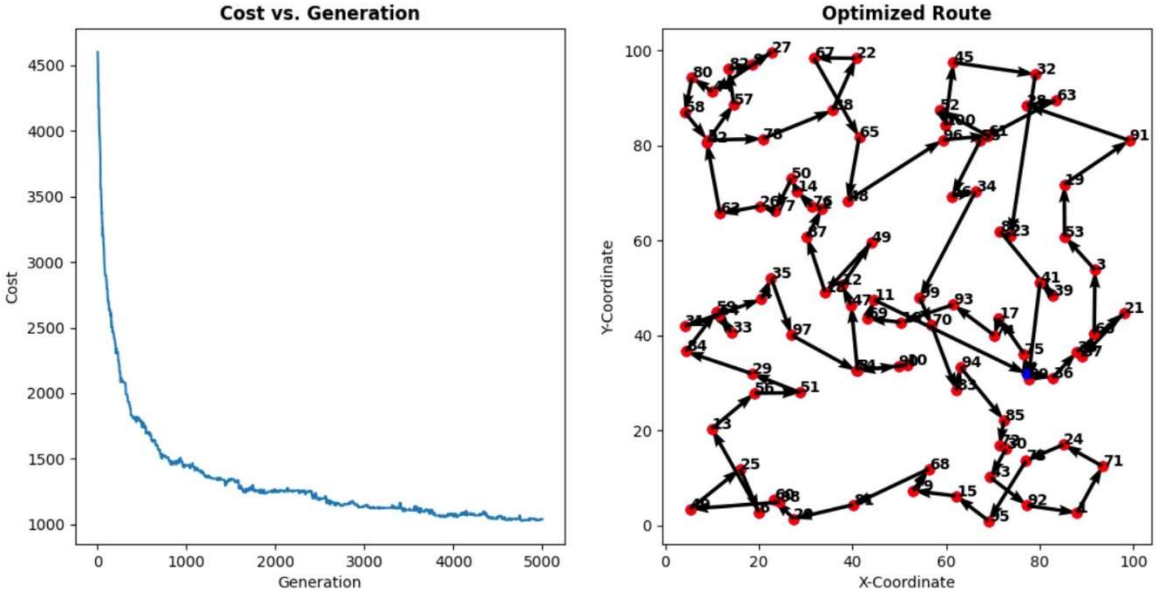
TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.001), Generations(3000)



TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.001), Generations(4000)



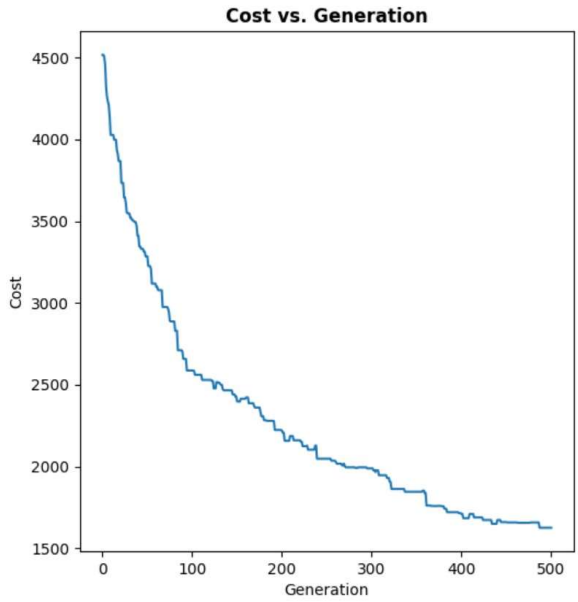
TSP Genetic Algorithm: Population(50), EliteNum(10), MutationRate(0.001), Generations(5000)

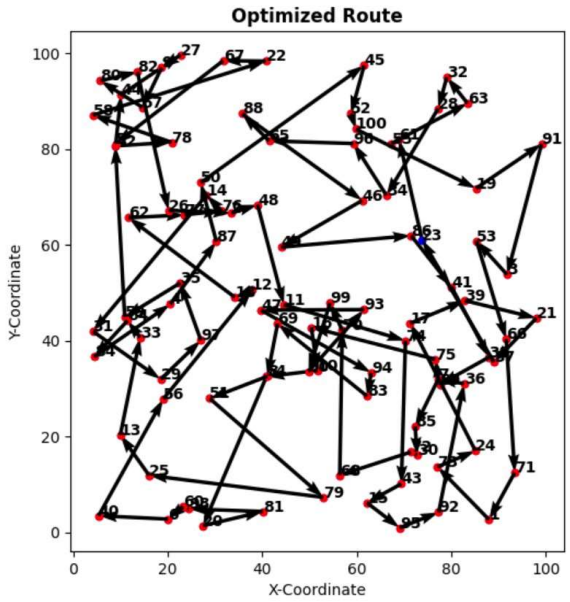



```
Final Dist: 1094.137242104103
```

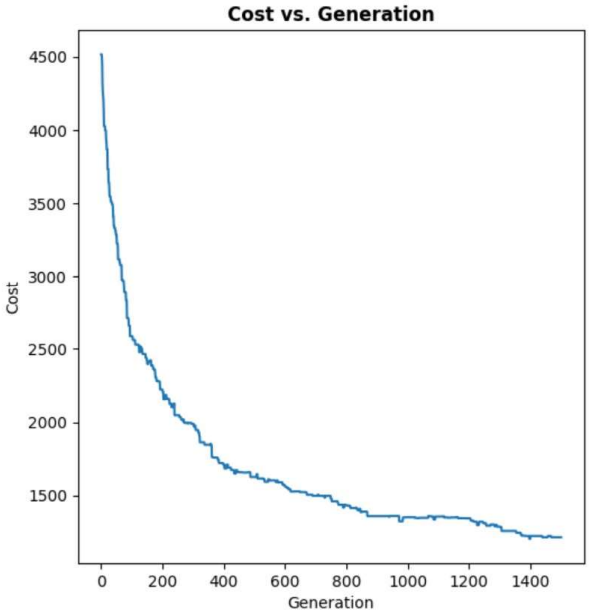
Dataset 2A

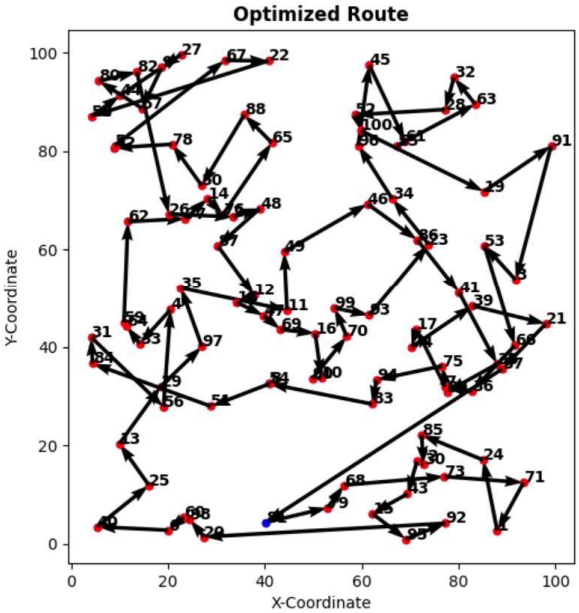
TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.0005), Generations(500)



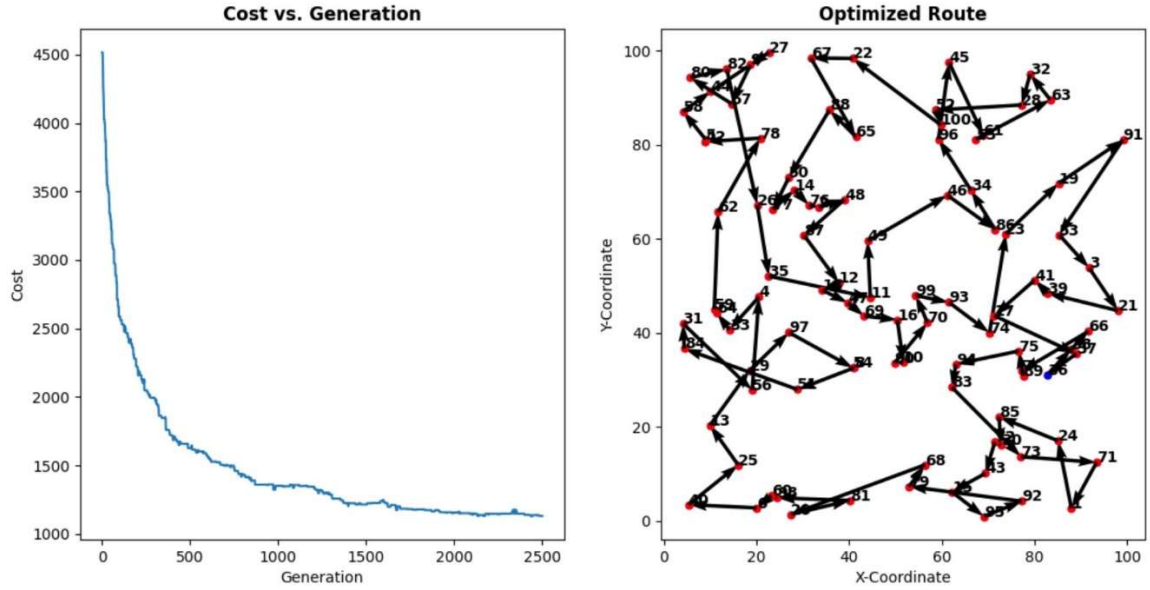


TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.0005), Generations(1500)

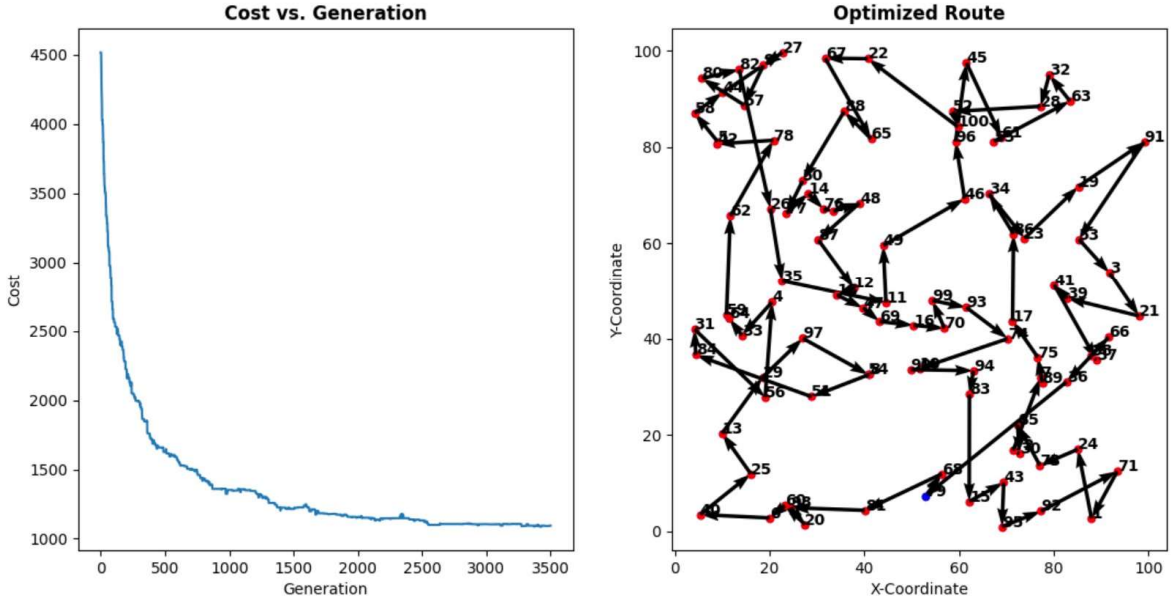




TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.0005), Generations(2500)



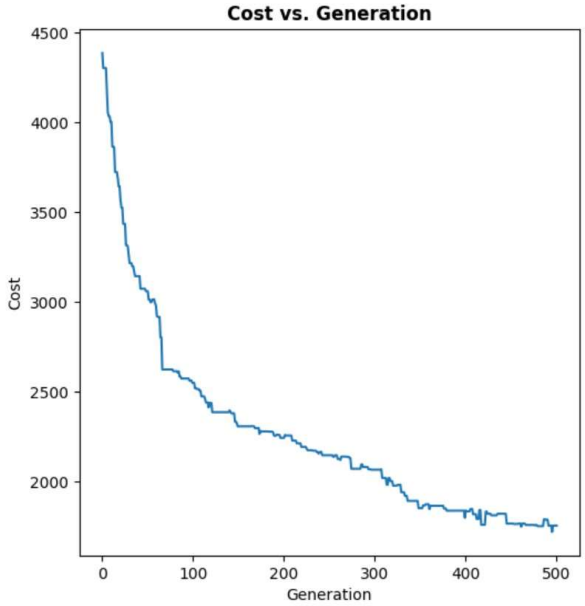
TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.0005), Generations(3500)

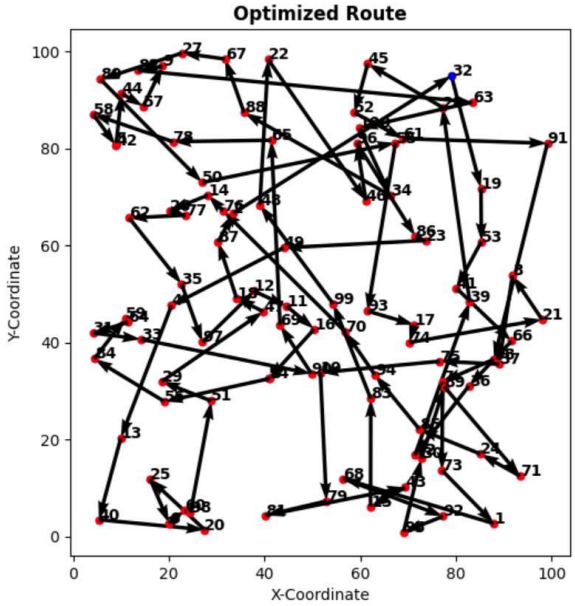



```
Final Dist: 1041.84008698355
```

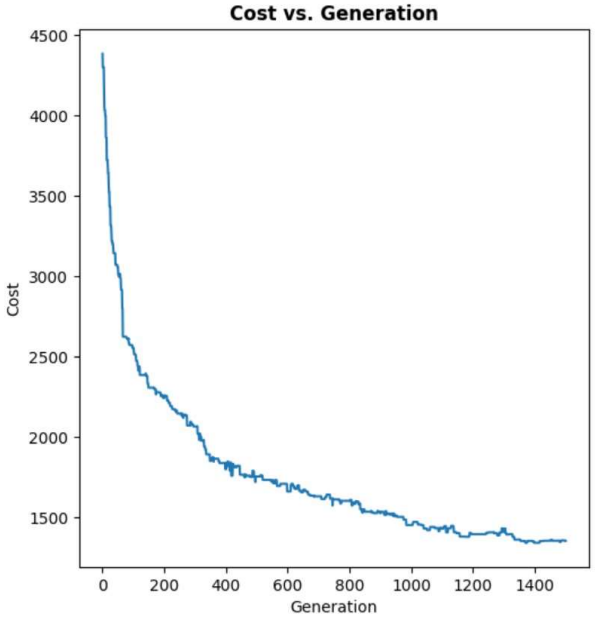
Dataset 2B

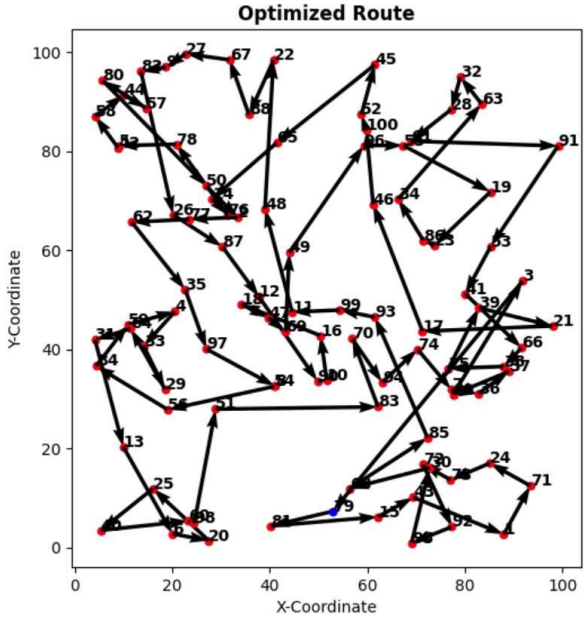
TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.001), Generations(500)



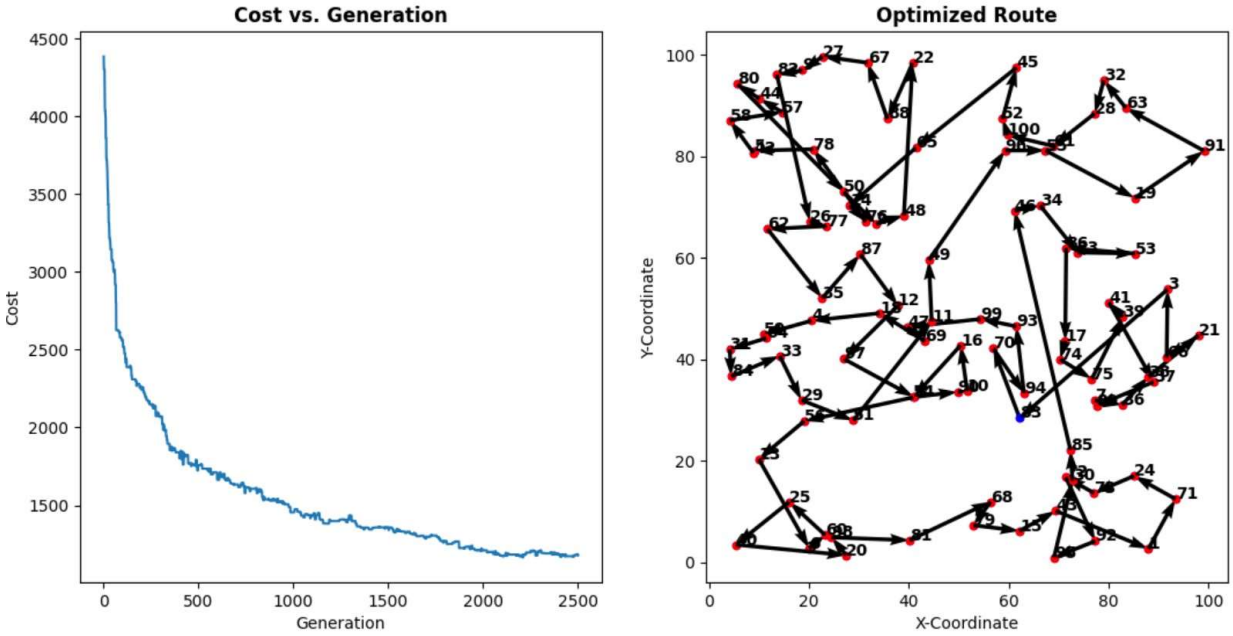


TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.001), Generations(1500)

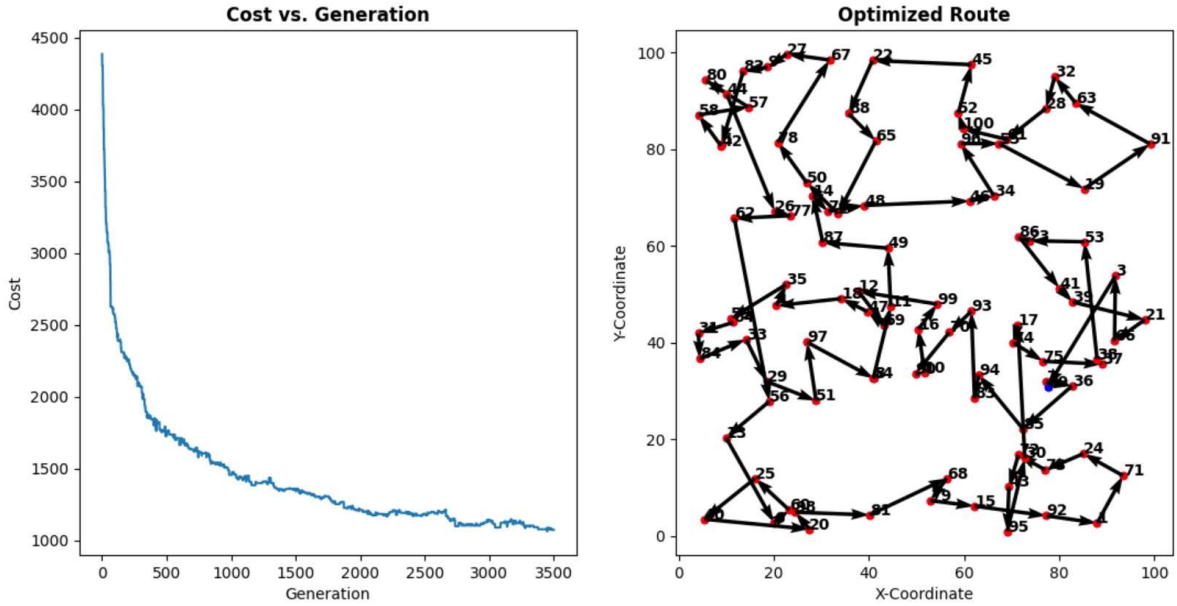


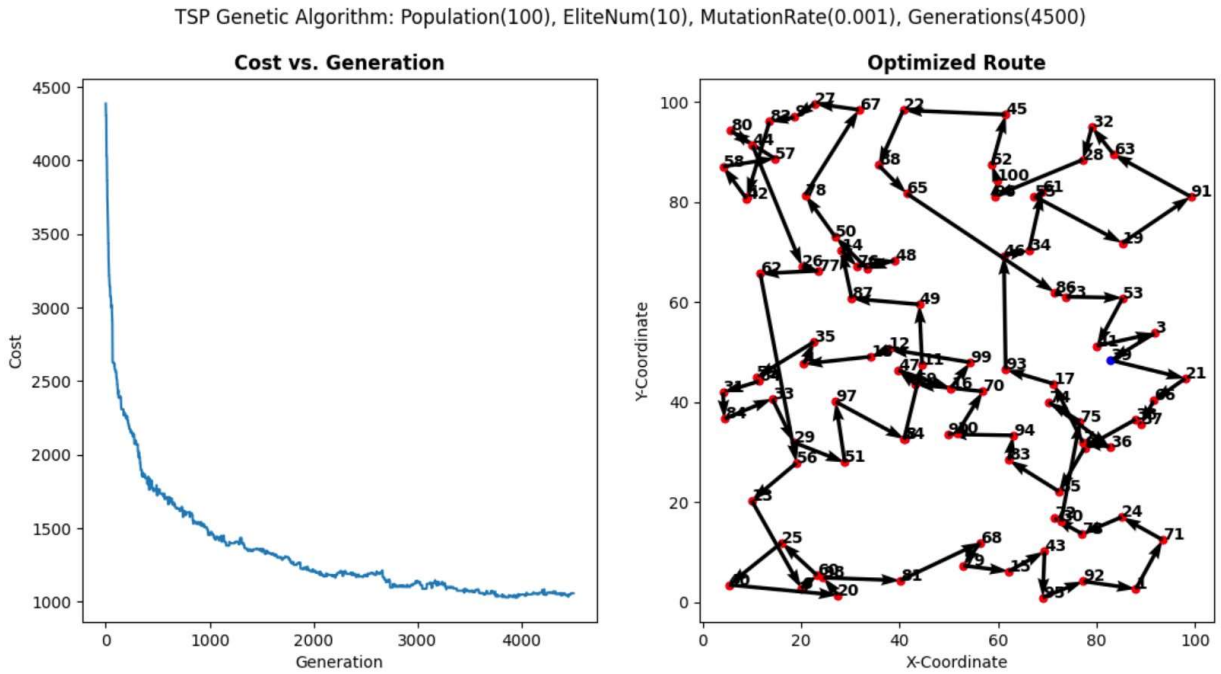


TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.001), Generations(2500)



TSP Genetic Algorithm: Population(100), EliteNum(10), MutationRate(0.001), Generations(3500)






```
Final Dist: 1067.018418217073
```

4. Discussion

Based on the results above, it can be concluded that Population Size and Mutation Rate as variable parameters do have an effect on the overall path cast after several generations. Given that population size refers to the number of possible individuals (routes), it is seen how a population of size 100 computed better results over the population size of 50 with the same mutation rate. This is due to the fact that there are more route (individual) possibilities in a population of size 100 than size 50. With a population of size 100, the average and the standard deviation as seen above are also better (cost-wise) when compared to a population of size 50. For the mutation rate, it is seen that a lower mutation rate performs slightly better than a higher one due to the fact that the individuals in the 'bred' population have a lower chance of being mutated, ultimately meaning that there is a more constant decrease in cost as the number of generations increase.

For the stopping criteria, the program checks averages of the cost for every 500 generations and compares it to the average of the cost for the past 500 generations. If the difference between the current 500 and the previous 500 generations is less than 20, then the program breaks the generations loop and outputs the last computed cost. If the difference is more than 20, then the program continues the generations loop and resets the counter to 0 for the next calculation of the following 500 generations. The max limit for the number of generations was set to be 8000. All of the runs above have fell under that limit where they have all flattened out with an average less than 50 as they got around to the 5000 generations mark. The code structure is shown below:

```
if (counter == 500):
    graphPlot(progress, listIdx, route)
    plt.clf()
    if (listIdx == 500):
        oldAvg = average(progress, 0)
        counter = 0
    else:
        avgByFar = average(progress, listIdx)
        if (abs(avgByFar - oldAvg) < 20):
            print("Avg too small")
            break
        else:
            oldAvg = average(progress, listIdx)
            counter = 0
```

Out of the four datasets presented above, the best solution was computed to be 1041.84009 with a population size of 100 and mutation rate of 0.0005, which is synonymous to our trend that was described above. Higher population size and a lower mutation rate usually output the most favorable results as there are a lot of possibilities for individual routes and the low mutation rate keeps the best performing individuals intact by reducing the probability of mutation on certain individuals.

For the algorithm runtime, population size of 50 with a mutation rate of 0.0005 took about 6-8 minutes while a mutation rate of 0.0001 took about 12-14 minutes. Likewise, population size of 100 with a mutation rate of 0.005 took about 15-18 minutes while a mutation rate of 0.001 took about 25-30 minutes. The lower the population size and mutation rate, the faster the algorithm runtime computed to be. This is expected because the program constructs the population as a list in which the size of the list is increased by double when population size is increased from 50 to 100. As seen in the improvement curves above for all four datasets, a mutation rate of 0.001 resulted in more generations to find the optimal cost over a rate of 0.0005. Therefore, the algorithm runtime presented is accurate and expected.

The biggest problem that I had faced in the implementation and analysis of this project was the runtime for each of the runs which became a hassle when trying to debug code. Especially with the specifications of my laptop, it became harder to run datasets consecutively in a row as the laptop would heat up and needed to restart after a while. Data collection for each of the runs also became progressively longer as the population size increased and each set of runs resulted in a new minimum. The redundancy in having to do so many runs/generations to compute best results was my biggest problem. One thing that I would change if I were to do this project again would be to pick different variable parameters like two different crossover methods instead of two different population sizes to see the effect it produces on the optimal paths. With the results above, it can be seen how there are several intersections (crossovers) in the paths presented which indicates that it isn't the most optimal solution. Therefore, I would want to see if other variable parameters have an effect in producing a more optimal path than the one presented above. Another thing that I wish I could change is the ability to run this code on a more powerful PC to accelerate runtime.

After completing this project, I have learned that the Genetic Algorithm is an efficient manner to replicate the behavior of nature and evolution for a computer-based problem. If I were to run this 100-city dataset using one of the earlier approached like Brute Force or BFS/DFS, it would take weeks, months, or even years. Using basic principles of reproduction, the algorithm uses the 'survival of the fittest' theory to sort out the best performing routes and eliminates the high-cost-producing ones. Given that it uses a stochastic search method, a lot of generations maintain a stable fitness until a superior chromosome kicks in. In the realm of the Traveling Salesman Problem, the Genetic Algorithm is a good approach to tackle larger datasets but hold its disadvantage of being randomized at the beginning throughout the end in terms of all possible chromosomes, mutations occurrence, crossovers, and more.

Overall, I believe that the Genetic Algorithm is a great approach to tackle larger datasets and produce near-to-optimal solutions depending on the parameters chosen. Given that it's faster than any of the other approaches that were explored in the past few projects, it is very advantageous in terms of runtime. Additionally, the algorithm's relevance to nature makes it easier to understand with basic principles of life and generate various interpretations to solve the problem. However, the random heuristics and incompleteness of the algorithm makes it difficult to arrive at an optimal path during some generations.

5. References

Matplotlib Documentation - https://matplotlib.org/stable/index.html
Numpy Documentation – https://numpy.org/doc/
Pandas Documentation – https://pandas.pydata.org/docs/
Genetic Algorithm Explained (TSP) - https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35