Traveling Salesman Problem (TSP) – Genetic Algorithm with Wisdom of Artificial Crowds

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 combination of a Genetic Algorithm and the Wisdom of Crowds approach 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 unenhanced Genetic Algorithm, Closest Edge Insertion, Breadth-First Search (BFS) & Depth-First Search (DFS), and Brute Force approaches that were explored in the last four 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

This project included a combination of two algorithms, the Genetic Algorithm (GA) and a Wisdom of Crowds (WOC) approach, to solve the Traveling Salesman Problem. Sections 2.1 and 2.2 go more in-depth about each approach. Majority of the code in this project was manipulated from Project 3 (Greedy) and Project 4 (GA). The WOC approach was applied after several runs of GA to really inspect the experts being produced and aggregate common opinions to produce a more optimal route, thereby enhancing base GA. In the case that the aggregated output from expert opinions didn't resemble a full TSP path, a greedy algorithm, the Closest Edge Insertion heuristic more specifically, was used to visit outlying cities.

2.1 Genetic Algorithm

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)

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.

2.2 Wisdom of Crowds

Now that the GA has been setup properly using the code from Project 4, there need to be several runs of the GA on a certain dataset with varying population, eliteNum, and mutationRate to create a diverse 'crowd' of experts that can help compute a more optimal route. More specifically, there were 4 unique runs of GA, with the above varying parameters, for every dataset that was provided. In the first two, the population size was set to be low which has a high chance of producing far-to-optimal solutions given the minimal diversity. On the contrary, the third and fourth runs of GA used a larger population size which meant that there would be more diversity within the individuals. Using the above information, the GA returns the top 5 fittest individuals for each of the first two runs while returning top 15 fittest individuals for the last two to generate a diverse expert 'crowd' for WOC to be applied upon. After each of the experts from every run have been extended onto a master expert crowd list, the program then enters the WOC aggregation algorithm to combine expert opinions.

In the WOC function, the main focus of combining/aggregating the expert opinions lies in the format of an n x n (n = # of cities) adjacency matrix that measures the occurrences of each edge within the expert 'crowd.' Using the automatic aggregation method of Yampolskiy et al. in their research paper, *Wisdom of Artificial Crowds – a Metaheuristic Algorithm for Optimization*, that focuses on producing a common solution after filtering through frequent local structures of the individuals within the expert crowd, the program first counts the number of occurrences for each edge present in the crowd and then removes the reversed duplicates of these tuples to provide more flexible options for the new route. With this aggregation method, it is believed that an optimal path will reflect the most popular edges in the expert crowd over less popular edges.

Once a list of common edges (without their reversed duplicates) and their occurrences within the crowd have been computed, the program then moves on to constructing a more optimal path by starting off with the most popular edge and re-iterating through the common edges list over and over again until it finds a match wherein either a connecting edge has edge[0] == startEdge[1] or edge[1] == startEdge[1]. Once it finds its next most popular connecting edge, that specific edge is then appended on to the lists of visitedEdges and visitedCities to re-iterate through the list of uncheckedEdges to find the next connection. This process is repeated until all the edges and cities have been checked and none of them are outlying. In the rare case that the path being produced even after the aggregation of expert opinions above does not resemble a valid TSP path, the program enters into a greedy algorithm structure wherein it utilizes the Closest Edge Insertion heuristic from Project 3 to break edges and visit every city that has not been covered by the aggregation algorithm above.

Finally, the WOC approach then outputs the final optimal path that it believes to be the same if not more optimal than the average being produced by the four GA runs. This above process for this project covers most of the techniques/algorithms (Brute Force, Greedy, and Genetic Algorithm) that have been covered in the past few projects to use portions of each to produce a grand optimal result.

3. Results

The enhanced GA algorithm gave near-to-optimal if not optimal solutions to the first few datasets but got progressively worse as the number of cities in each dataset increased. The route graphs for the last three datasets show several intersections (crossovers), especially the 222 City dataset, which shows us how the path is not optimal enough and could be lower. This is due to the randomness of the algorithm and personal computer capacity/speed which is discussed later on.

3.1 Data

The data being inputted was provided in the form of .tsp files which had included a list of cities and their corresponding x and y coordinates. More specifically, there were six different provided datasets with the coordinates of 11, 22, 44, 77, 97, and 222 different cities respectively. Each 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 (11 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

11-City Dataset			
Run	GA	GA AVG	WOC
Run 1	351.04588	352.0027	351.0459
Run 2	351.04588		
Run 3	354.87316		
Run 4	351.04588		

77-City Dataset			
Run	GA	GA AVG	WOC
Run 1	1126.841	973.8007	834.1373
Run 2	1055.929		
Run 3	865.0538		
Run 4	847.3792		

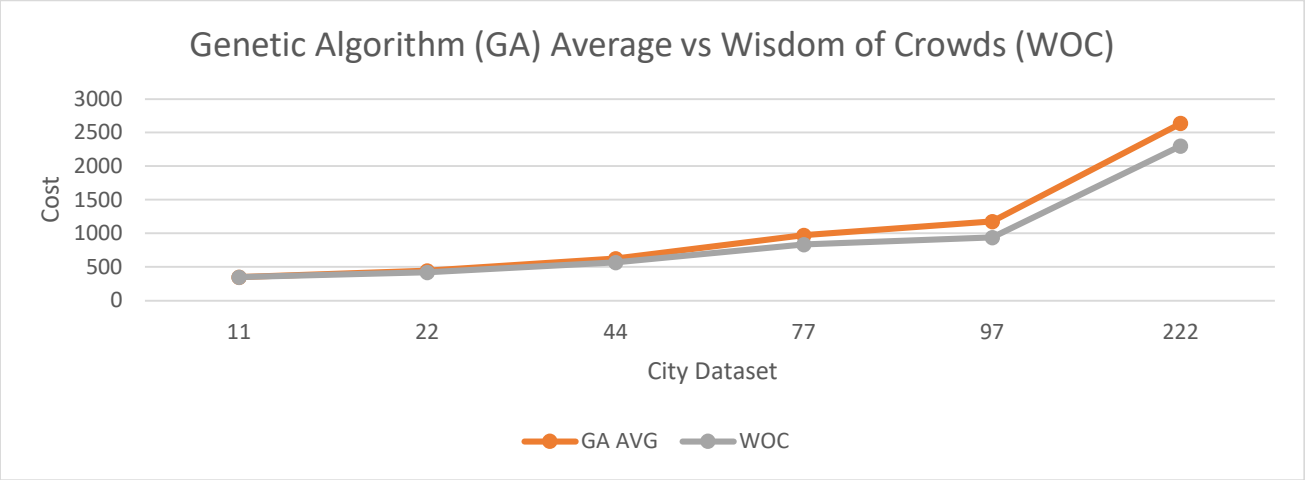
22-City Dataset			
Run	GA	GA AVG	WOC
Run 1	443.0441	443.0762	419.2035
Run 2	490.85369		
Run 3	419.20348		
Run 4	419.20348		

97-City Dataset			
Run	GA	GA AVG	WOC
Run 1	1367.652	1178.358	939.8205
Run 2	1266.826		
Run 3	1110.848		
Run 4	968.1069		

44-City Dataset			
Run	GA	GA AVG	WOC
Run 1	684.42972	625.1028	567.7049
Run 2	630.22481		
Run 3	589.04092		
Run 4	596.71573		

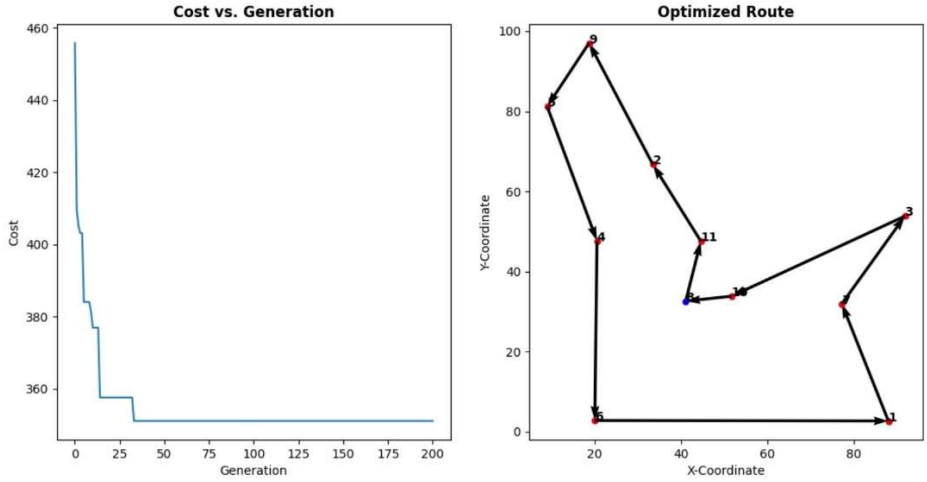
222-City Dataset			
Run	GA	GA AVG	WOC
Run 1	2836.676	2639.297	2302.597
Run 2	2906.233		
Run 3	2308.342		
Run 4	2505.936		

Dataset	GA AVG	WOC
11	352.0027	351.0459
22	443.07619	419.2035
44	625.10279	567.7049
77	973.80073	834.1373
97	1178.3583	939.8205
222	2639.297	2302.597

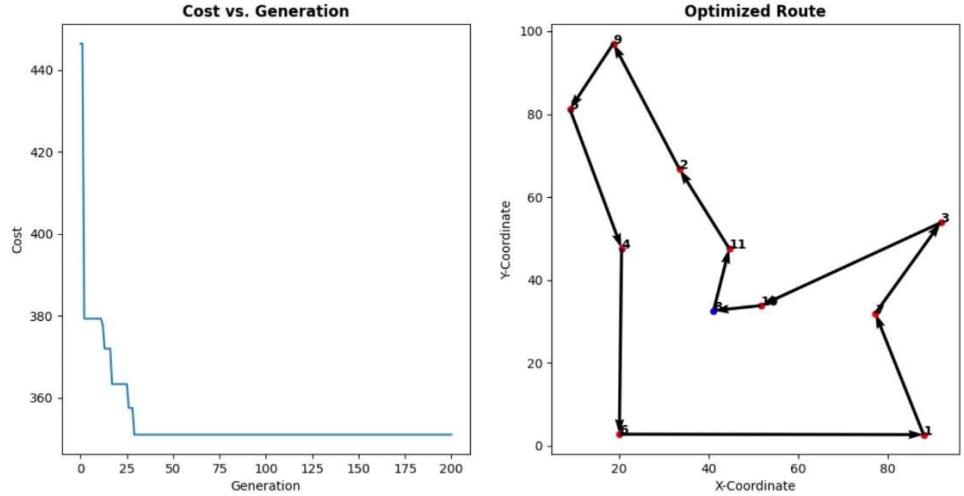


11-City Dataset

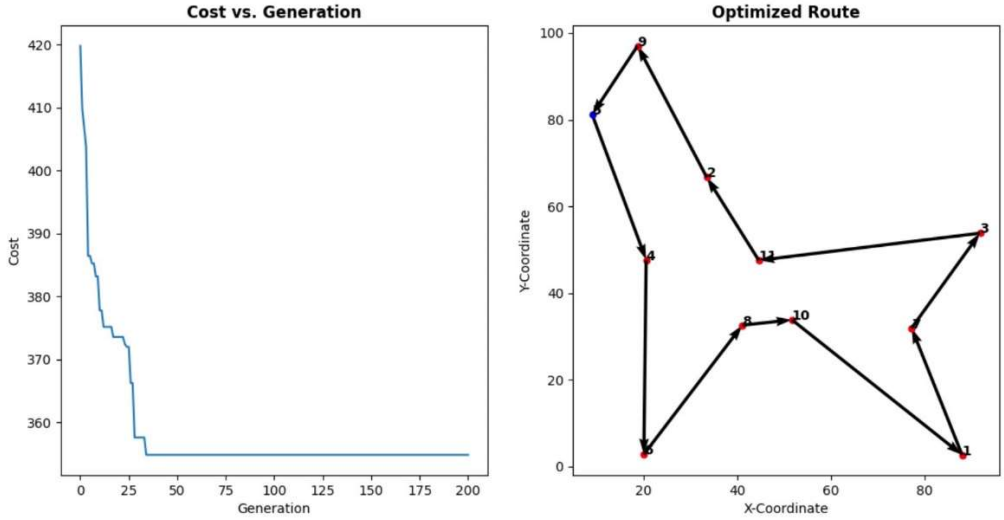
TSP Genetic Algorithm (RUN 1): Population(50) EliteNum(5) MutationRate(0.002) Generations(200), Distance(351.04587994184993)



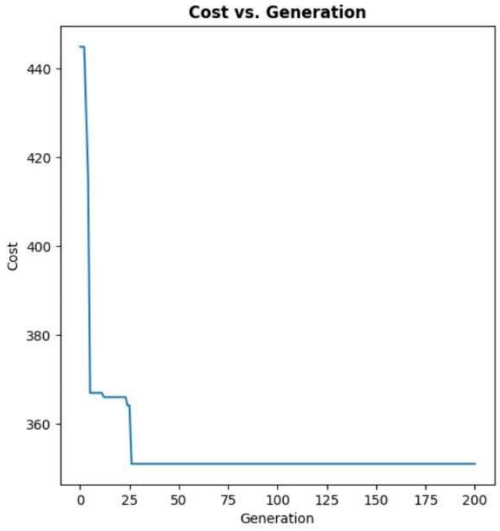
TSP Genetic Algorithm (RUN 2): Population(75) EliteNum(10) MutationRate(0.002) Generations(200), Distance(351.04587994184993)

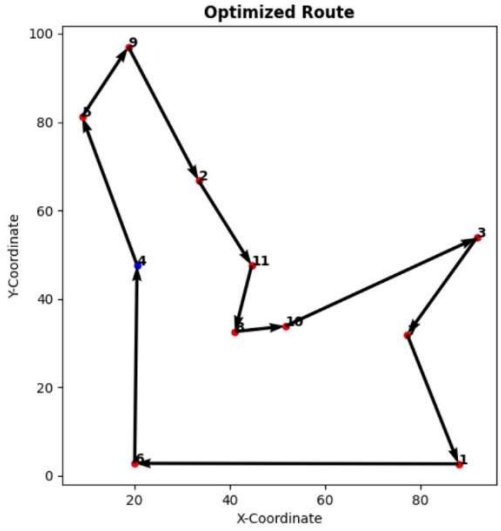


TSP Genetic Algorithm (RUN 3): Population(100) EliteNum(15) MutationRate(0.002) Generations(200), Distance(354.8731614418321)

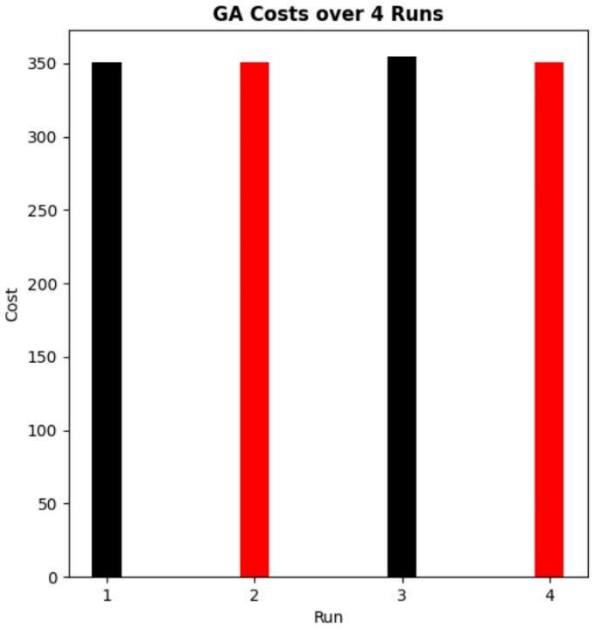


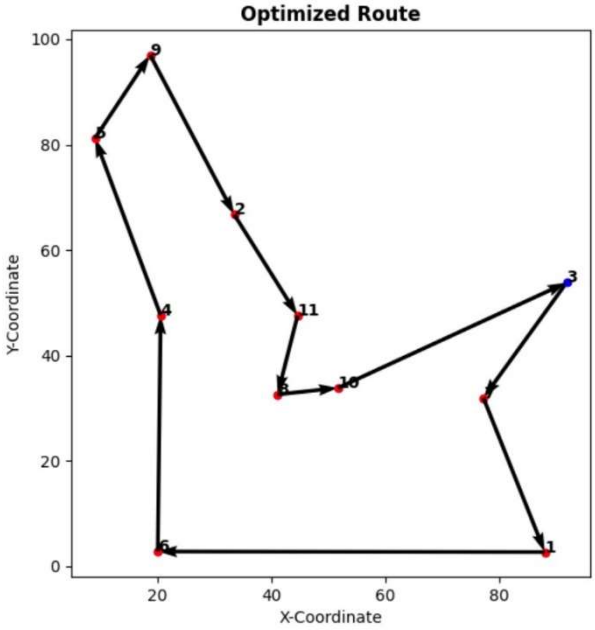
TSP Genetic Algorithm (RUN 4): Population(125) EliteNum(20) MutationRate(0.002) Generations(200), Distance(351.04587994184993)





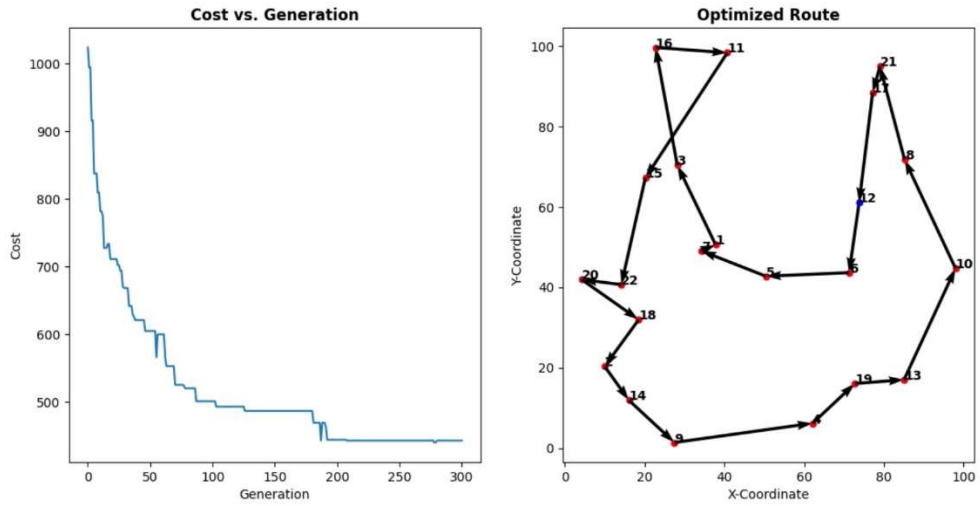
TSP GA + WOC Results: Optimal WOC Solution = 351.04587994185



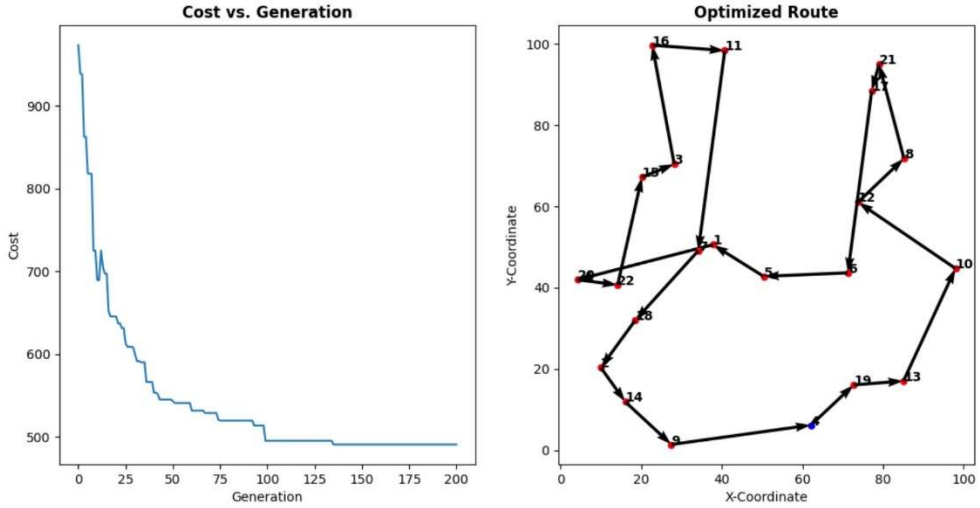


22-City Dataset

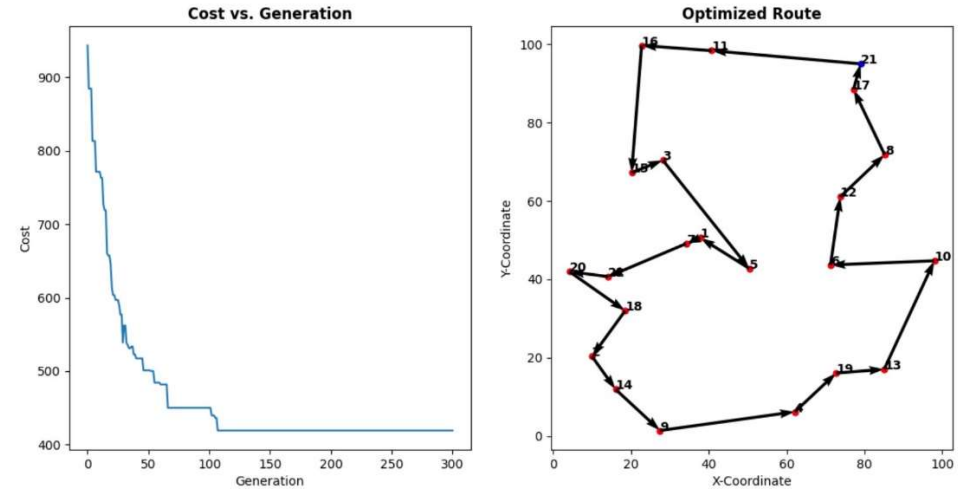
TSP Genetic Algorithm (RUN 1): Population(50) EliteNum(5) MutationRate(0.002) Generations(300), Distance(443.04410460359725)



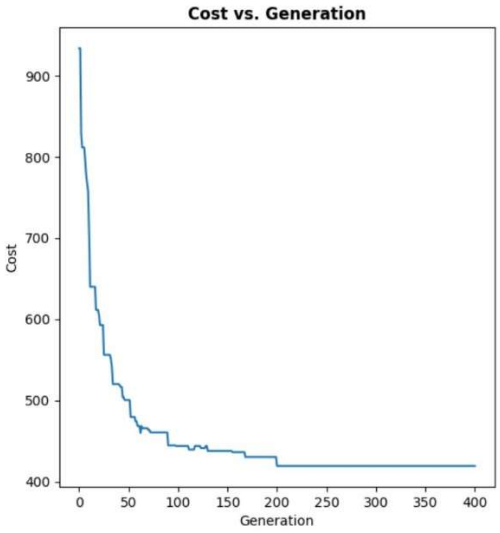
TSP Genetic Algorithm (RUN 2): Population(75) EliteNum(10) MutationRate(0.002) Generations(200), Distance(490.85369257429306)

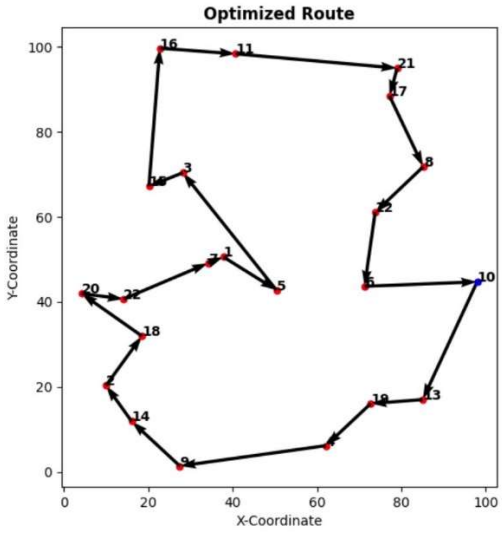


TSP Genetic Algorithm (RUN 3): Population(100) EliteNum(15) MutationRate(0.002) Generations(300), Distance(419.2034766399671)

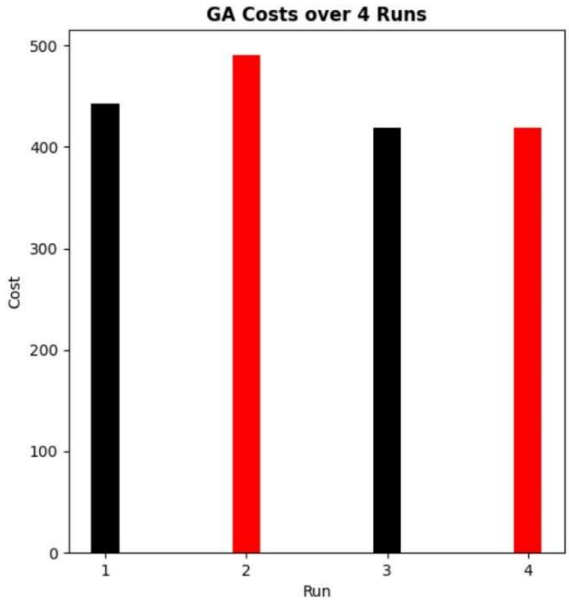


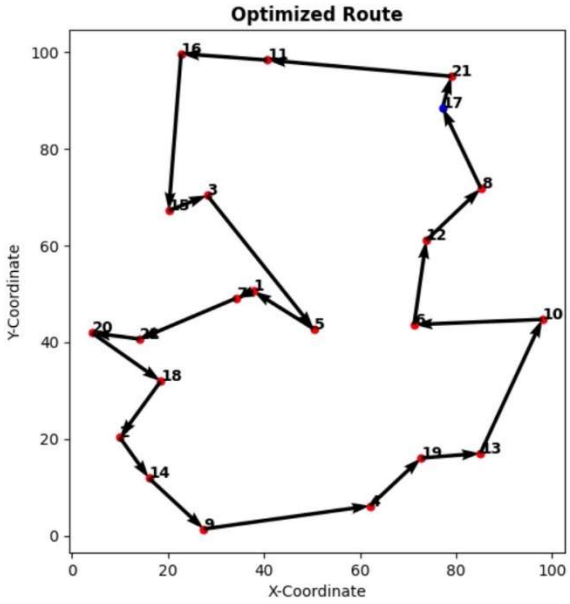
TSP Genetic Algorithm (RUN 4): Population(125) EliteNum(20) MutationRate(0.002) Generations(400), Distance(419.2034766399671)





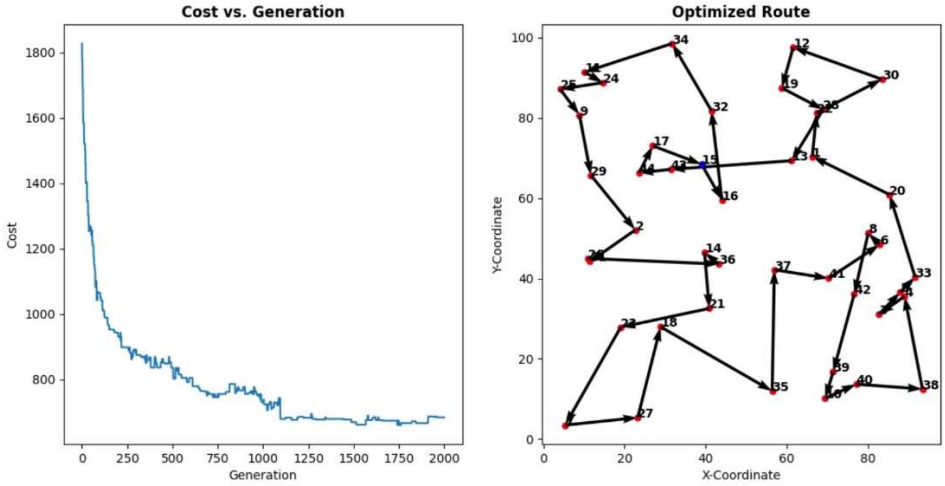
TSP GA + WOC Results: Optimal WOC Solution = 419.2034766399671



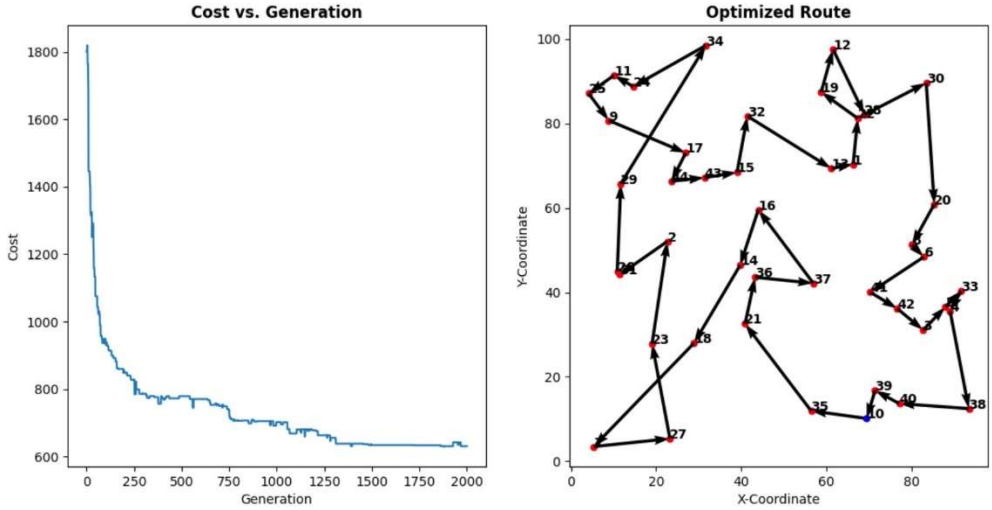


44-City Dataset

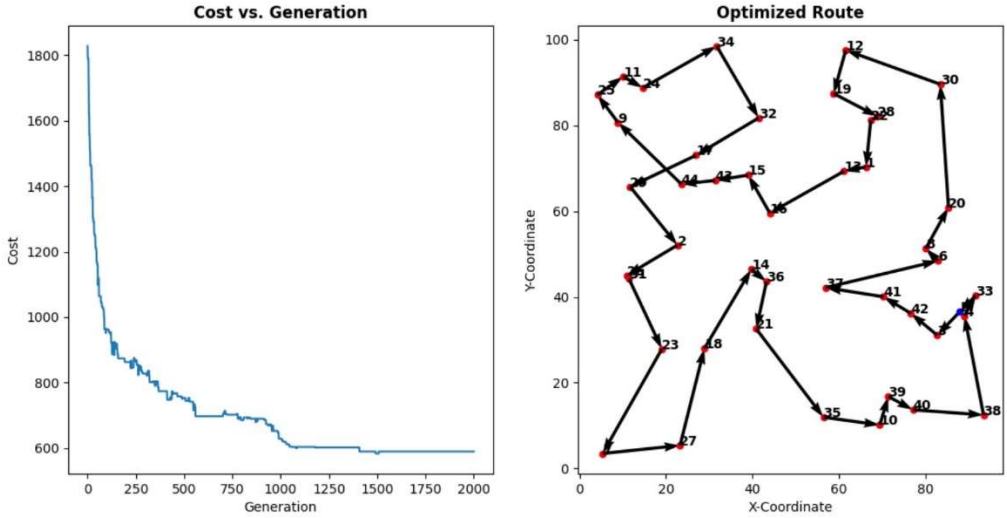
TSP Genetic Algorithm (RUN 1): Population(50) EliteNum(5) MutationRate(0.002) Generations(2000), Distance(684.4297169604222)



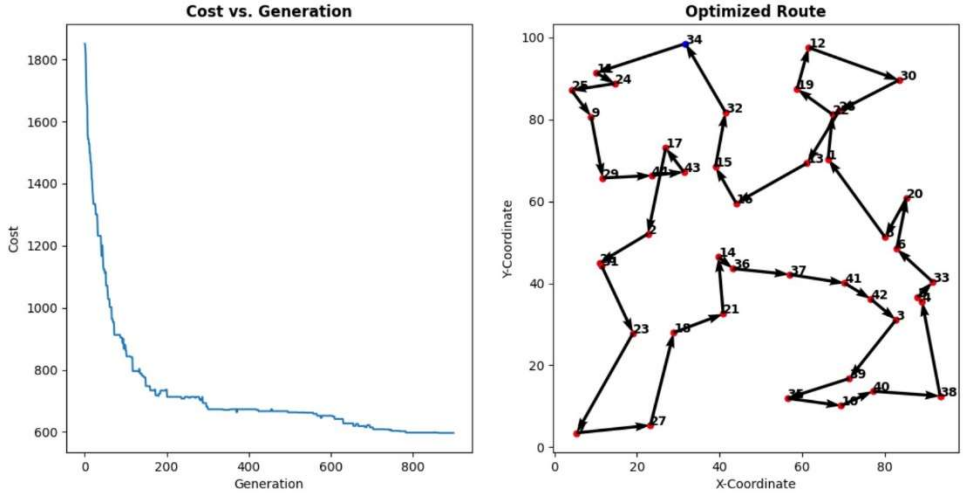
TSP Genetic Algorithm (RUN 2): Population(75) EliteNum(10) MutationRate(0.002) Generations(2000), Distance(630.224811230957)



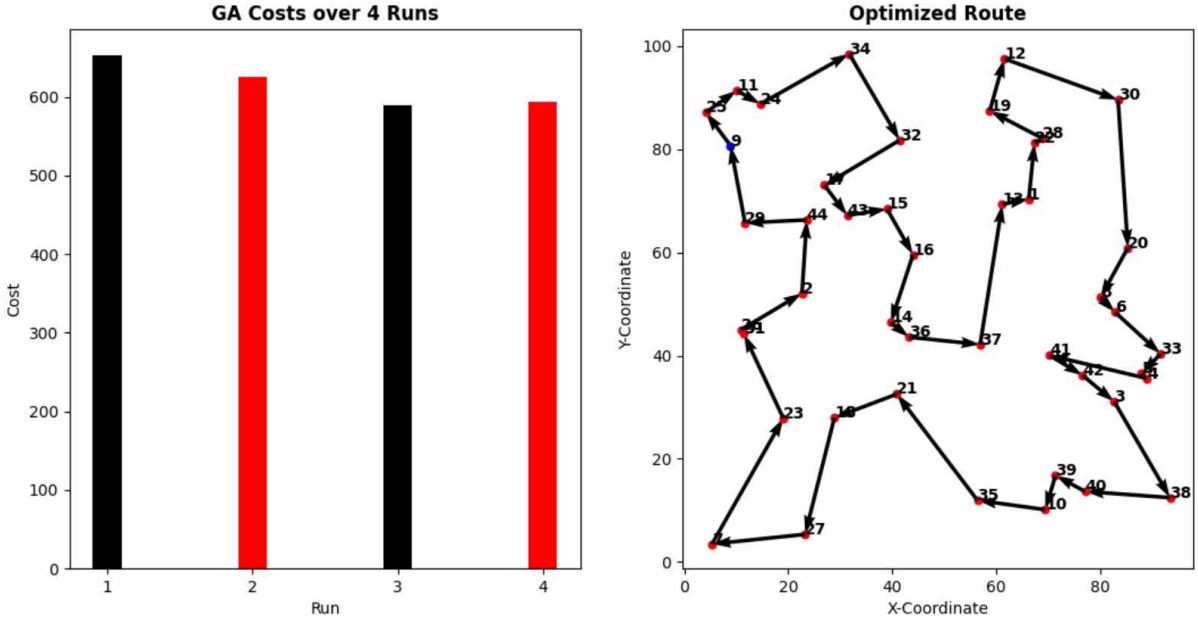
TSP Genetic Algorithm (RUN 3): Population(100) EliteNum(15) MutationRate(0.002) Generations(2000), Distance(589.0409233953251)



TSP Genetic Algorithm (RUN 4): Population(125) EliteNum(20) MutationRate(0.002) Generations(900), Distance(596.7157283773374)

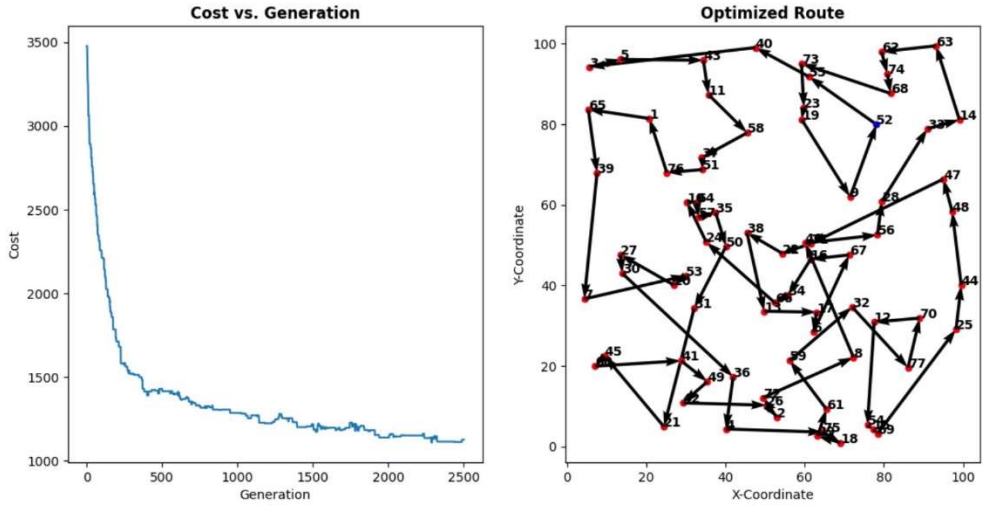


TSP GA + WOC Results: Optimal WOC Solution = 567.7048793457177

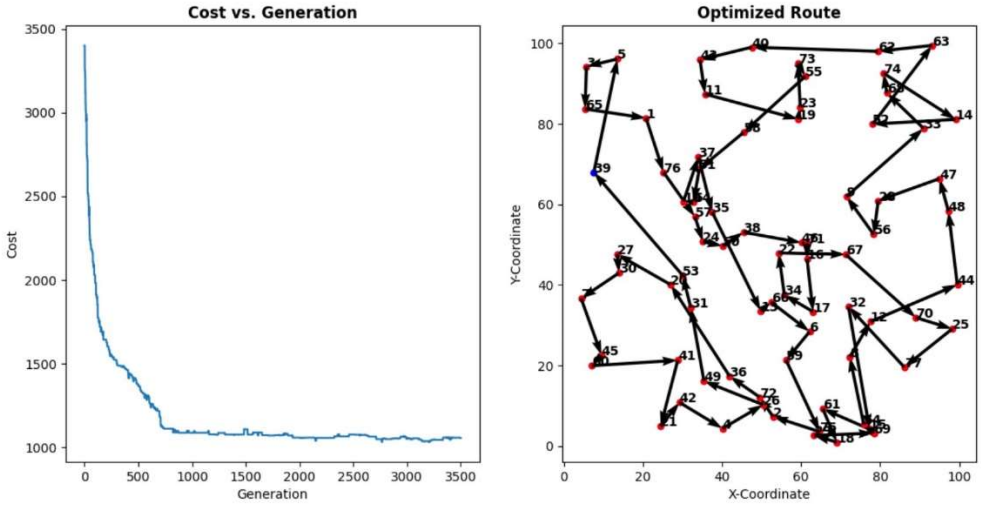


77-City Dataset

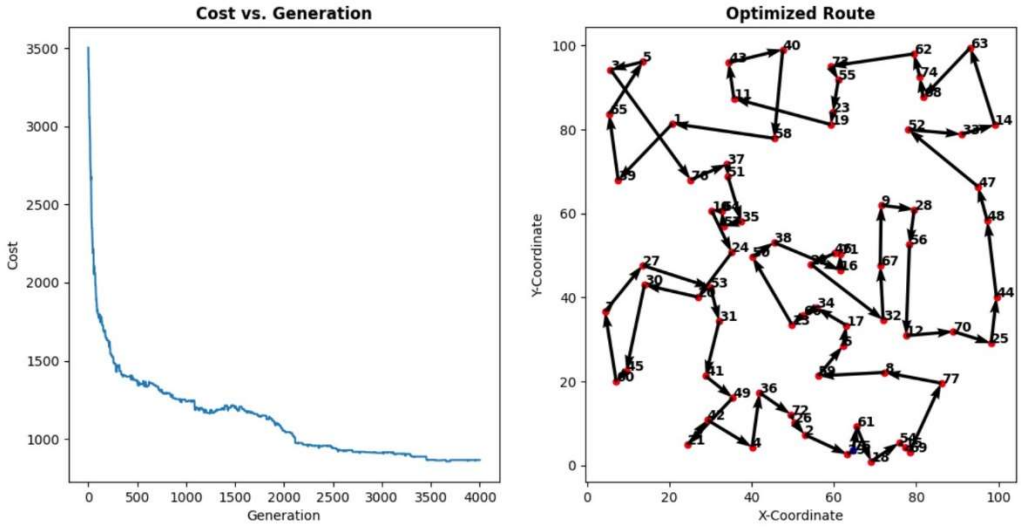
TSP Genetic Algorithm (RUN 1): Population(50) EliteNum(5) MutationRate(0.0008) Generations(2500), Distance(1126.8406667509432)



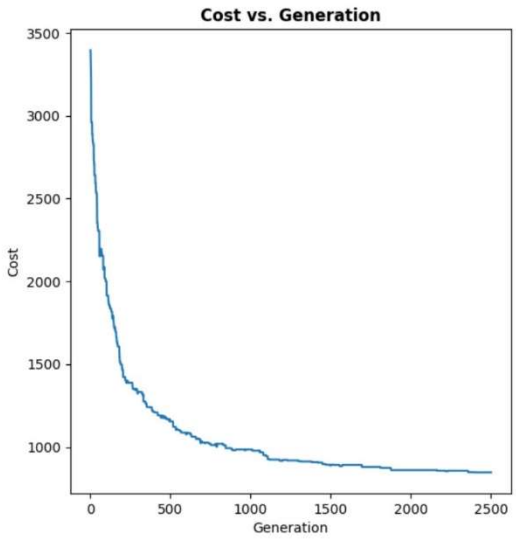
TSP Genetic Algorithm (RUN 2): Population(75) EliteNum(10) MutationRate(0.0008) Generations(3500), Distance(1055.9292137687653)

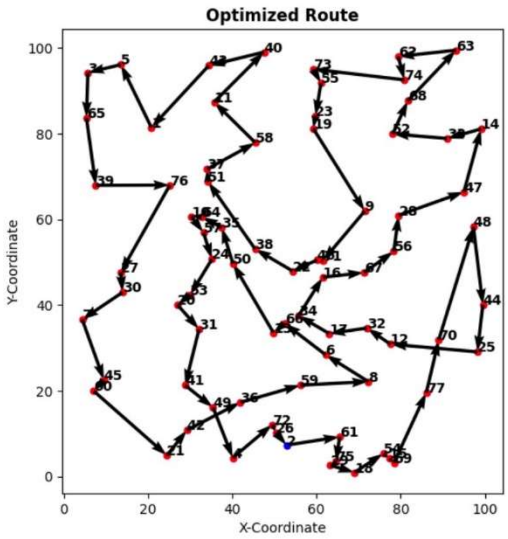


TSP Genetic Algorithm (RUN 3): Population(100) EliteNum(15) MutationRate(0.0008) Generations(4000), Distance(865.0538229603942)

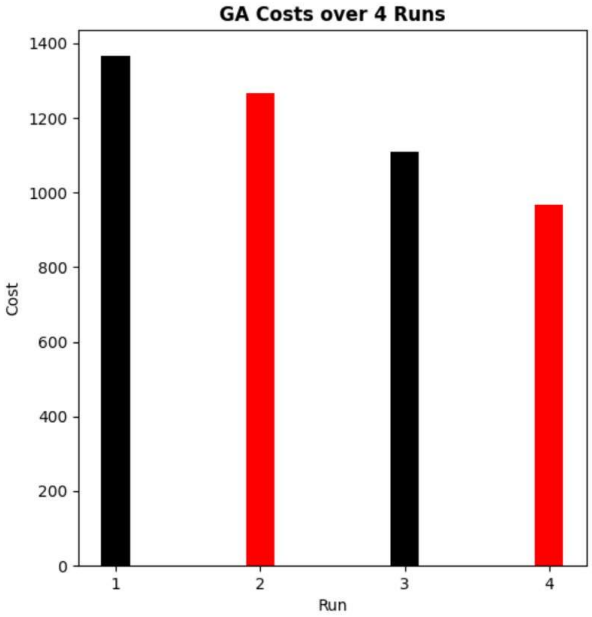


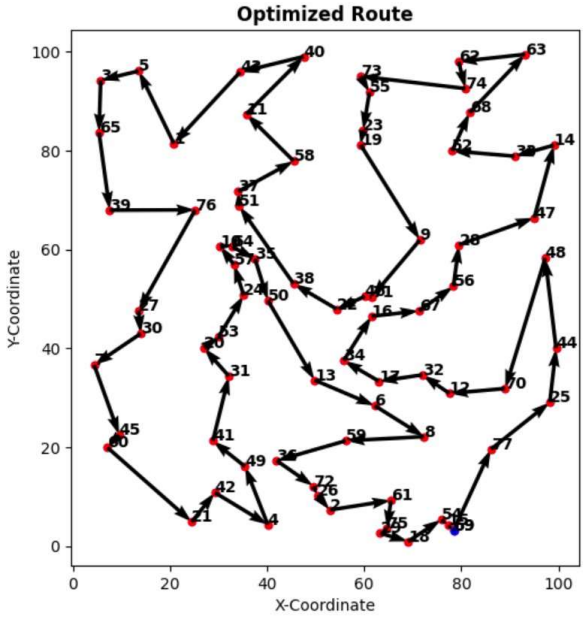
TSP Genetic Algorithm (RUN 4): Population(125) EliteNum(20) MutationRate(0.0008) Generations(2500), Distance(847.3792227246322)





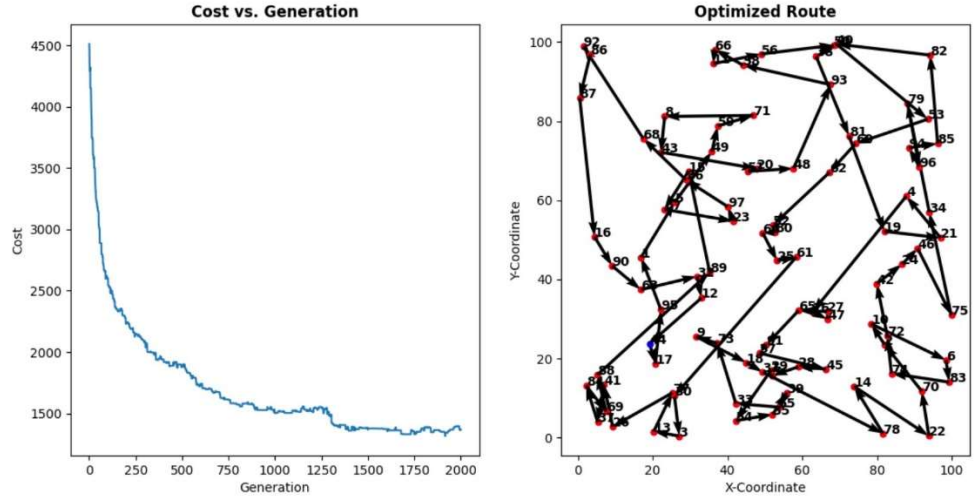
TSP GA + WOC Results: Optimal WOC Solution = 834.1372817959802



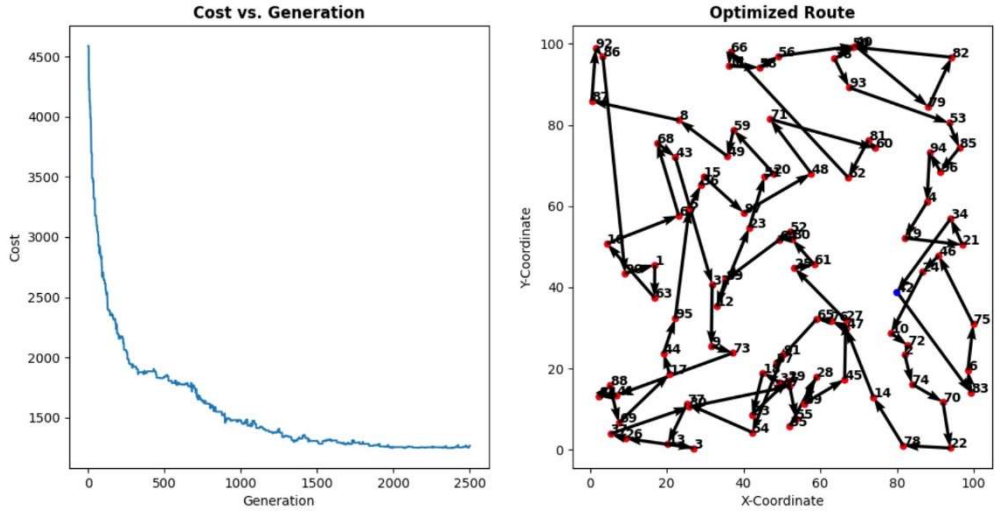


97-City Dataset

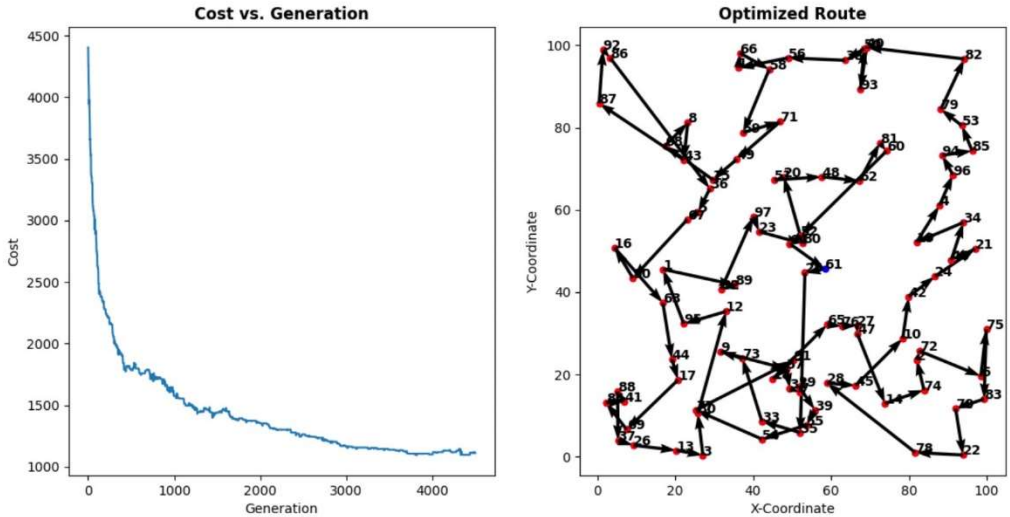
TSP Genetic Algorithm (RUN 1): Population(100) EliteNum(10) MutationRate(0.0008) Generations(2000), Distance(1367.6515714754316)



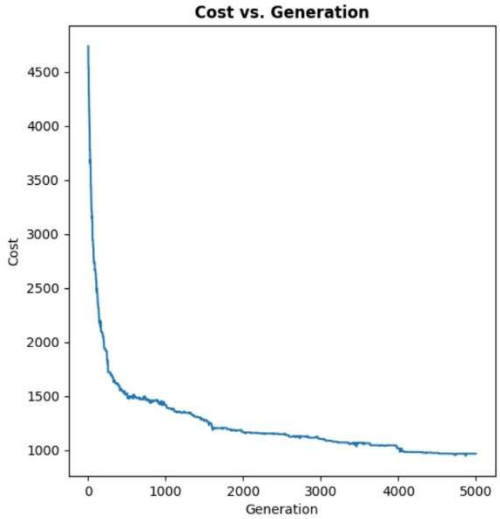
TSP Genetic Algorithm (RUN 2): Population(125) EliteNum(15) MutationRate(0.0008) Generations(2500), Distance(1266.8264419481632)

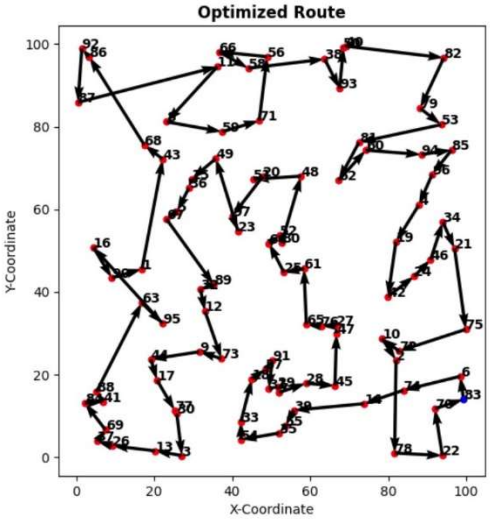


TSP Genetic Algorithm (RUN 3): Population(100) EliteNum(10) MutationRate(0.0008) Generations(4500), Distance(1110.848435314757)

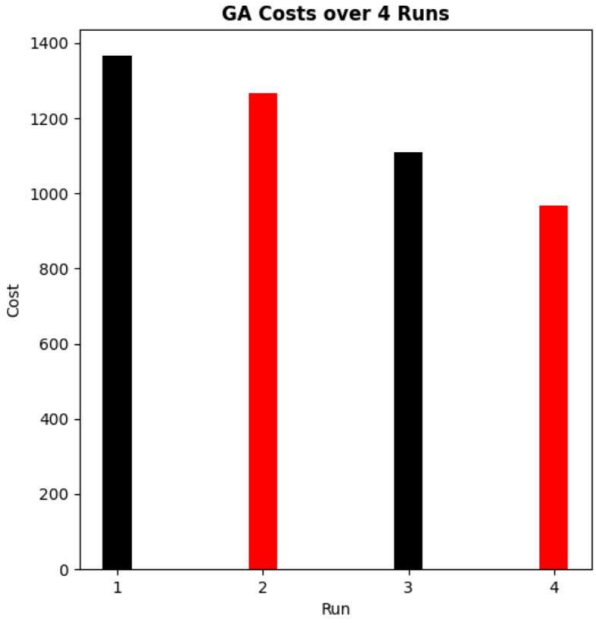


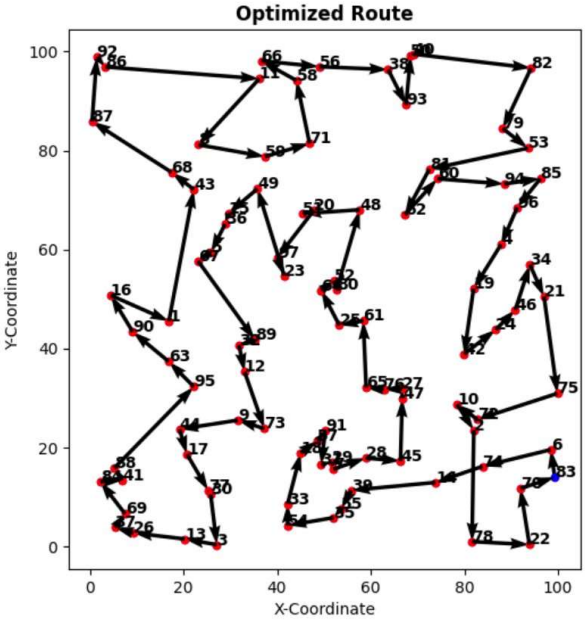
TSP Genetic Algorithm (RUN 4): Population(125) EliteNum(15) MutationRate(0.0008) Generations(5000), Distance(968.1069083066776)





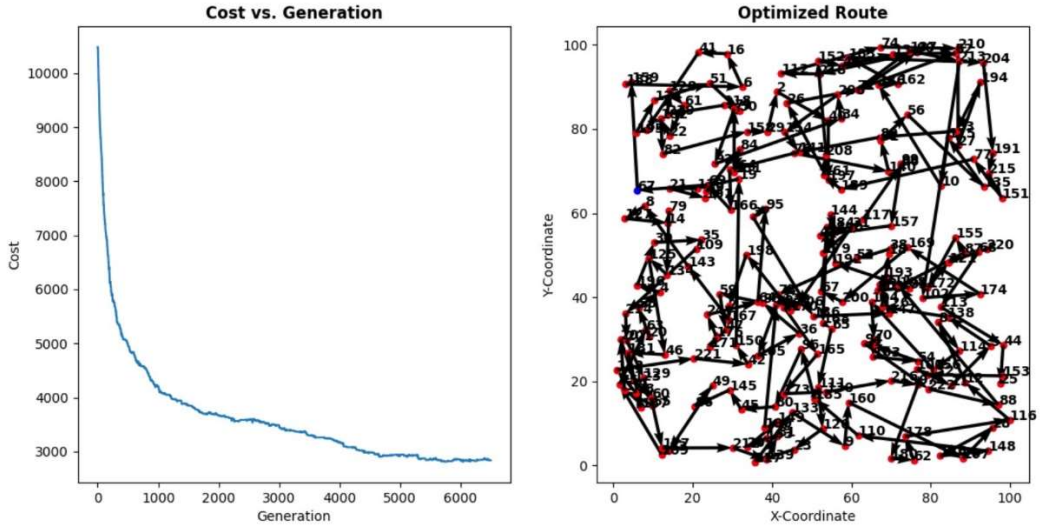
TSP GA + WOC Results: Optimal WOC Solution = 939.8204621268213



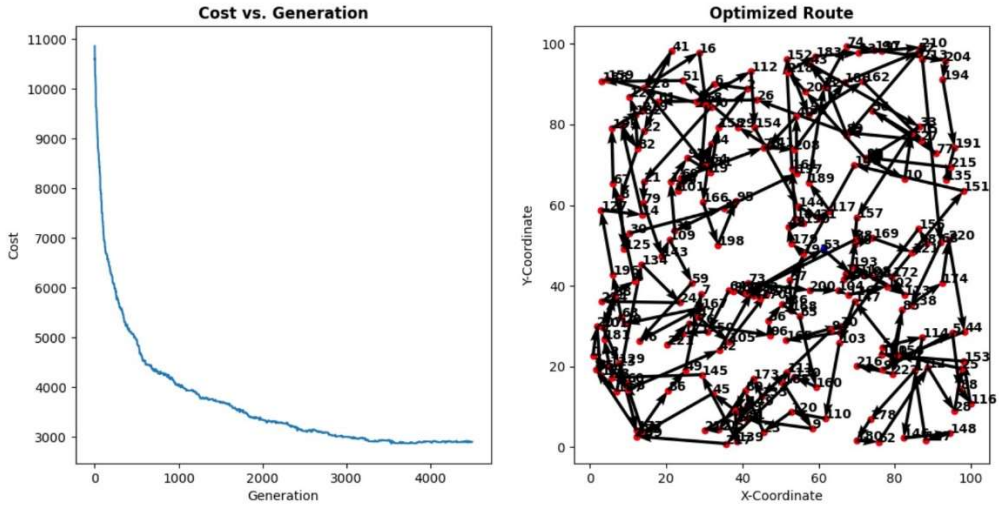


222-City Dataset

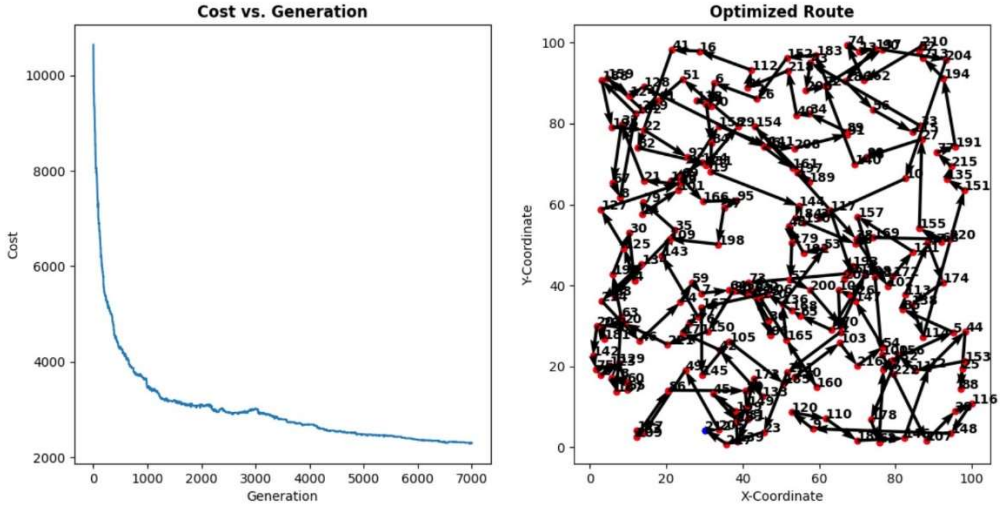
TSP Genetic Algorithm (RUN 1): Population(40) EliteNum(8) MutationRate(0.0002) Generations(6500), Distance(2836.6755784938337)

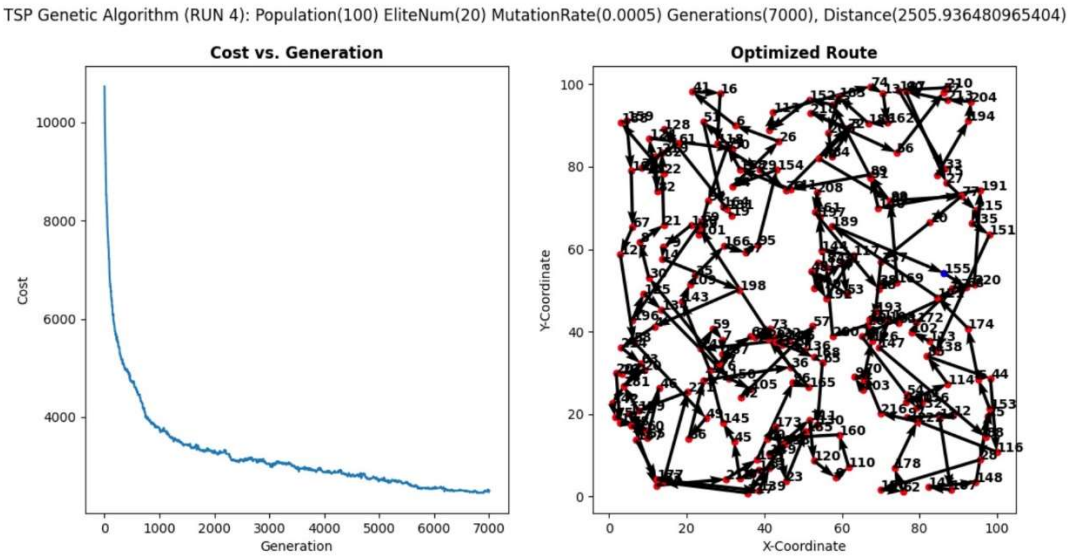


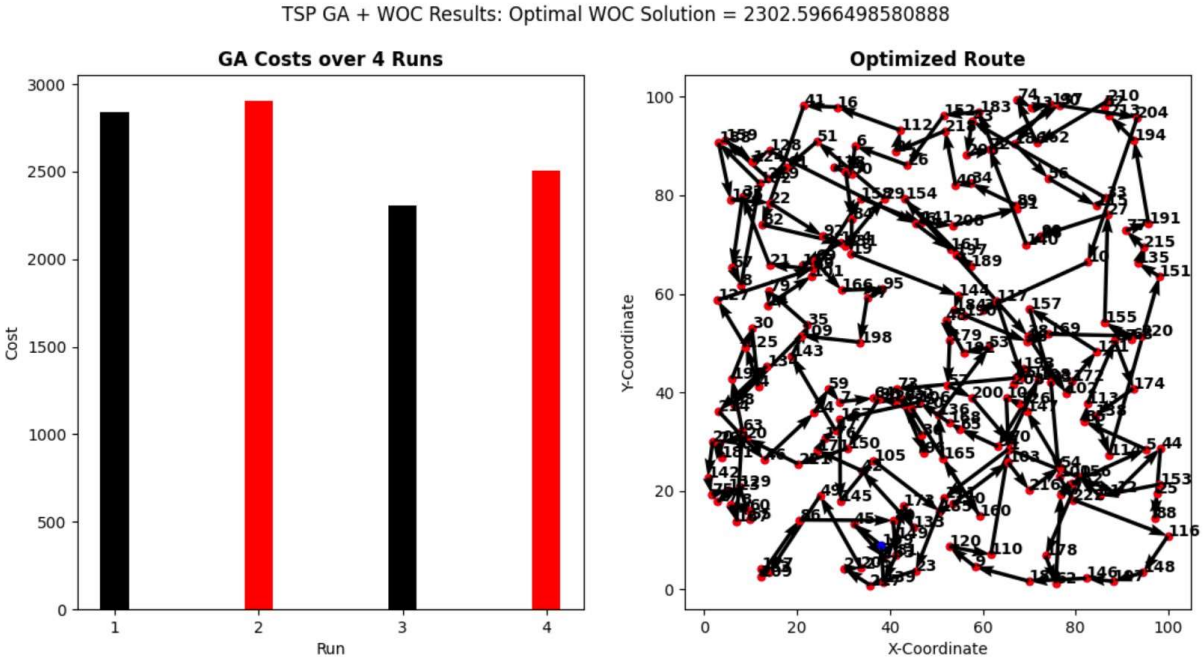
TSP Genetic Algorithm (RUN 2): Population(60) EliteNum(12) MutationRate(0.0003) Generations(4500), Distance(2906.2334260890834)



TSP Genetic Algorithm (RUN 3): Population(80) EliteNum(16) MutationRate(0.0004) Generations(7000), Distance(2308.342403070574)







4. Discussion

Based on the results and charts above, it can be seen how the Wisdom of Crowds approach, on average, performs the same if not better than the best GA Run output's cost. Although there were several crossovers being produced in the optimal solution as the number of cities in the datasets increased, the WOC approach and the aggregation method chosen above proved to provide a similar or more optimal solution for each dataset. It was very rare that the path generated by the WOC approach, and the aggregation method had to be carried into the greedy algorithm to make it a valid TSP solution. It occurred during some instances where the expert crowd had a lot of varying routes which made it difficult to find common edges. This problem mainly occurred in the 97-City and 222-City datasets because of the huge ranges of expert individuals given the low parameters being passed for efficiency on the

personal laptop. However, all the other datasets, given the size and parameters sent, visited every city with the aggregation method and did not have to enter a greedy heuristic. The genetic algorithm had a stopping criterion for each of the runs wherein 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 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
```

For algorithm runtime, performance, or speed of the above approach, the bigger datasets took a huge amount of time especially by running GA 4 times first and then applying WOC. Using estimation, here are the following total runtime measures for each dataset:

1) 11-City Dataset ≈ 4 minutes
2) 22-City Dataset ≈ 10 minutes
3) 44-City Dataset ≈ 18 minutes
4) 77-City Dataset ≈ 34 minutes
5) 97-City Dataset ≈ 48 minutes
6) 222-City Dataset ≈ 1 hour 15 minutes

The times above were estimated using real-world startTime and endTime because calculating it in the program accounted for the extra time needed to screenshot each of the GA outputs while the program was running. As seen, the runtimes get progressively worse as the number of cities increase. In the program above, several parameters were changed to speed up the process of the output to allow time for debugging and proper outputs. 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. If this same program were to be run on a much faster PC than a personal laptop, then the times would be significantly better and would allow for bigger population sizes.

The route graphs after applying WOC above show that the size of the problem (number of cities in a dataset) definitely does make a difference when computing for more optimal routes. For the first few datasets, there appears to be zero crossovers and multiple GA runs with the same costs. On the contrary, the GA runs have a stark variety of costs and multiple crossovers towards the end of the datasets with a larger number of cities. The largest that was tested in this project was the 222-city dataset with minimal parameters. If this program were to be tested on a more powerful computer, then the population size could've been increased which will allow for more diversity of individuals and possibly more optimal routes. The default randomness nature of the GA makes it difficult to come up with fit individuals with low population sizes, especially when the number of cities is very high. Therefore, a higher population size for the last three datasets running on a more powerful PC could significantly improve the results shown above.

After completing this project, I have learned that the combination of a Wisdom of Crowds approach and a 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 222-city dataset using one of the earlier approaches 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. The Wisdom of Crowds approach then takes the expert individuals of each GA run and creates an expert 'crowd' out of which a more optimal path can be produced. This additional WOC approach just enhances the base GA and outputs slightly better results over several GA runs.

Overall, I believe that the Wisdom of Crowds approach enhances the base Genetic Algorithm to tackle much larger datasets and produce near-to-optimal solutions depending on the parameters chosen. However, the random heuristics and incompleteness of the algorithm makes it difficult to arrive at an optimal path during some generations especially when dealing with lower population sizes on a personal laptop.

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
Yampolskiy, Roman V et al. "Wisdom of Artificial Crowds—A Metaheuristic Algorithm for Optimization." *Journal of Intelligent Learning Systems and Applications* 4 (2012): 98-107.
S. K. M. Yi, M. Steyvers, M. D. Lee and M. Dry, Wisdom of the Crowds in Traveling Salesman Problems, Available at: socsci.uci.edu/~mdlee/YiEtAl2010.pdf, Jan. 7, 2011.