# Python Implementation of Genetic Algorithm for TSP

***IE555: Course Project Report***

Dennis Posheluk and Felipe Meneguzzo Pasquali
*University at Buffalo*

## I. Introduction

The traveling salesman problem is a a route planning problem, in which the goal is to find the shortest (time or distance) possible route for a set of locations when the salesman must visit each city only once and then return to the origin location. Applications of the traveling salesman problem are widespread ranging from the routing of delivery trucks to tool path planning for the fabrication of microchips.

Heuristics algorithms can be used to solve TSP problems and provide robust solutions. In the IE555 course, a TSP problem (practice_25) based in the Buffalo region was solved using the nearest neighbor and simulated annealing heuristics. The practice_25 problem has twent five locations distributed in the Buffalo region as shown in Fig. 1. The salesman is driving from location to location using roadways. For the IE555 course project, the problem is solved using the Genetic Algorithm, an algorithm based on the principle of natural selection. The Python code has capabilities to solve for both minimal driving distance and driving time, but only minimizing distance is considered in this analysis.

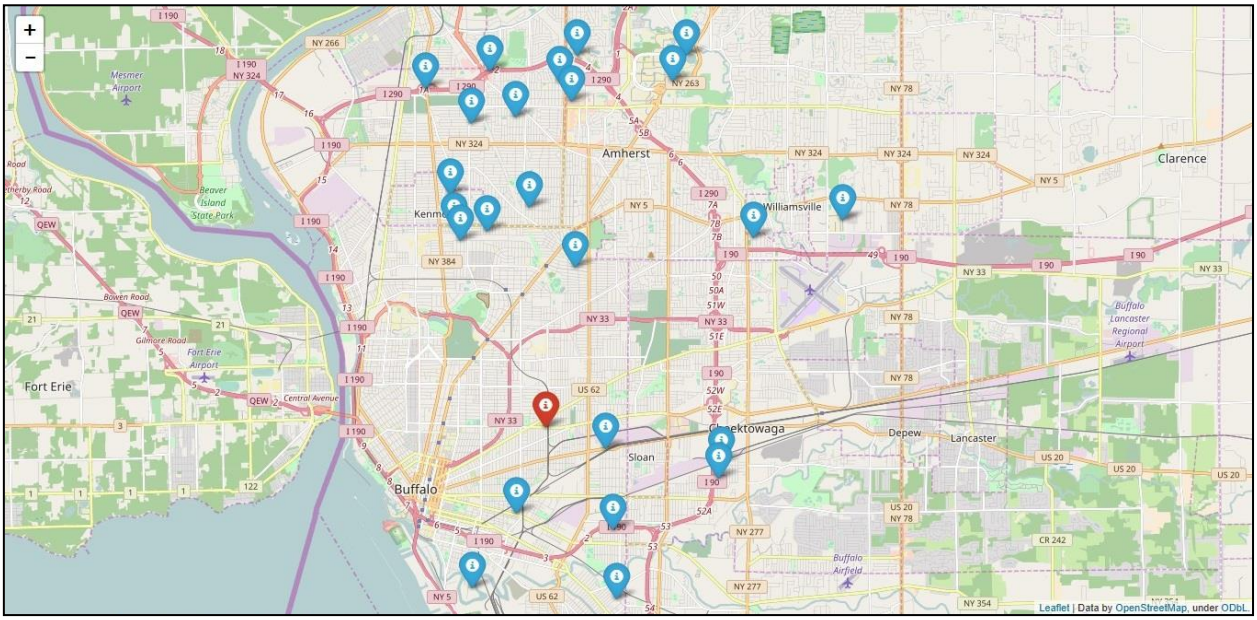


*Figure 1: Location Distribution of practice_25 Problem*

## II. Deliverables Overview

### 1. GA Python Code

A Genetic Algorithm capability is added to the TSP solving python code which has been developed throughout the semester in IE555. With the addition of GA, the user will have the ability to choose between nearest neighbor, simulated annealing, genetic algorithm, or MILP Gurobi to solve a TSP problem. The broad coding of the Genetic Algorithm is based on "Genetic algorithms for the traveling salesman problem" by Jean-Yves Potvin [1] with the PMX cross-over method, which is discussed in better detail in "Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation" by Gokturk Ucoluk [2].

### 2. Python Animations for GA

An animation of the convergence of the Genetic Algorithm is plotted for a single run. In the plot, for each iteration a new objective value appears representing the optimal objective value that iteration. Furthermore, a route evolution animation is also made to show the change in optimal route as the algorithm progresses through iterations.

### 3. practice_25 Problem GA Solution

The practice_25 problem is solved in Python using the Genetic Algorithm. The algorithm is run 10 times to provided data for analysis. The objective value (distance) and route sequence are provided every run. For further analysis, the GA solution is compared to SA and NN neighbor heuristic solutions.

### 4. Code Validation

To validate the python implementation of the Genetic Algorithm, the results obtained will be compared to results from the Gurobi solution.

## III. Methodology

### 1. GA Python Code

The python code uses the following libraries with the corresponding functionality [3]:

- sys: provide access to variables used by the interpretor
- csv: read and write in csv format
- folium: data visualization on leaflet map
- math: mathematical functions
- random: pseudo random number generators
- urllib2: Opening URLs
- json: data encoder and decoder
- pandas: data structures
- matplotlib.pyplot: data visualization
- matplotlib.animation: data animation visualization generation

- time: time tracking
- gurobipy: Gurobi optimizer for python

The portion of the code which reads the locations and generates the distance is carried over from earlier work in the course. Using command line inputs, the code accepts the location folder and which optimization methodology to use. The code imports the coordinates from the tbl_locations.csv file. With the location coordinates, the MapQuest API is used to import travel time and travel distance data for vehicle travel from MapQuest. The data is organized into pandas dataframes decoded by the json library.

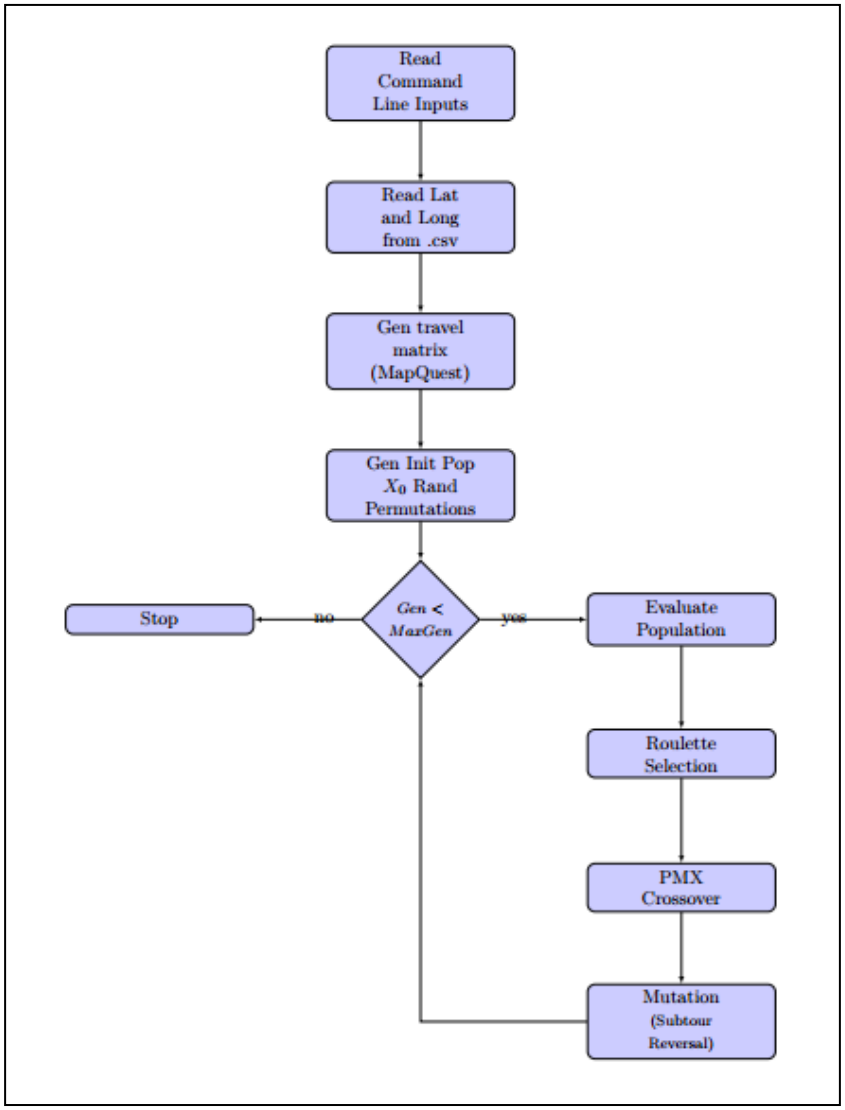


*Figure 2: Genetic Algorithm Flow Chart*

The GA algorithm, as shown in Fig. 2, is implemented by first generating an initial population through random permutations. The fitness of each member of the population is computed by calculating the tour cost (total distance) using data from the MapQuest API. Parents for the next generation are selected using fitness proportionate roulette wheel selection, where the probability of an individual to be selected as a parent is the fitness of that individual relative to the total fitness of the population. For the generation of the next population, the rate of crossover is .75, indicating that in 75 in every 100 "children" solutions will be cross-over products of their parents. The other 25 children will maintain their parents route. To undergo crossover, the potential child must have a randomly generated number lower than .75. The crossover is done using PMX crossover methodology. In PMX, for child 1, for each entry in the route sequence of parent 1, that location swaps positions with the value in it's own sequence which corresponds to the value of the entry in the sequence of the second parent. The same operation occurs vise versa for the second child of the pairing. For visual representation of the process, refer to Fig. 3 sourced from "Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation" by Gokturk Ucoluk [2].

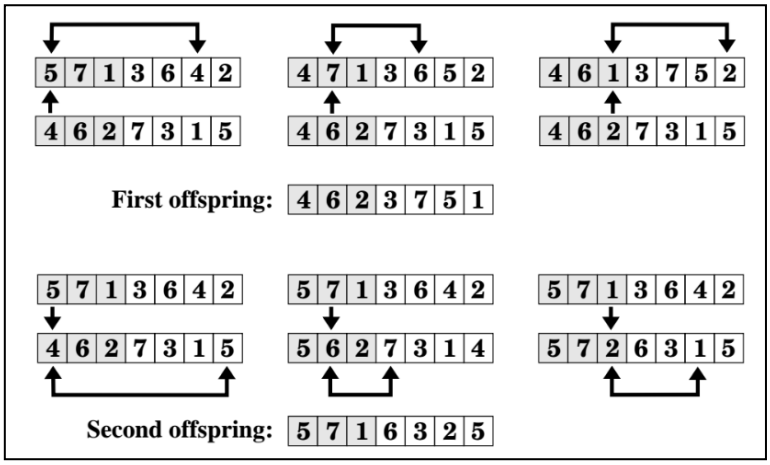


*Figure 3: PMX Crossover*

Once the crossover is complete, the population is mutated following a probability defined as mutation rate. The mutation consists in applying a subtour reversal in the randomly chosen individual.This process is then repeated for a total of number of specified iterations.

When the GA is completed, the optimal route sequence is plotted using folium onto a leaflet map. The starting point is represented by a different color than the other locations. The specific route lines based on roads to take are generated from the Mapquest API.

**2. Python Animations for GA**

Two animations are generated for use in the project video, a plot of the convergence for each iteration, and also a visualization in route changes as the algorithm iterates. Initially, the

approach was to record a live animation of the two visuals, but the algorithm had a run time of ~30 minutes, making the video not applicable to the 10 minute video.

As an alternative, at each progressing iteration the best objective value , and best route sequence were appended to a list. After all the iterations were completed, the imported matplotlib.animation library is used to generate animations. The animation function works by iteratively updating the plot for the specified number of iterations. The animation were plotted on the same plot as sub-plots to show the comparison of objective value and route at each iteration. The animation for the route was generated using matplotlib rather than folium, because a folium animation feature for this application was not discovered. The Euclidean plot of the route generated as shown in figure 4, is not representative of the actual roads that will be taken, but rather it is meant to be a representation of the sequence of locations. Furthermore ffmpeg, a multimedia processing software, was downloaded to save to animation. The animation library has a save function which works with ffmpeg as a video writer.

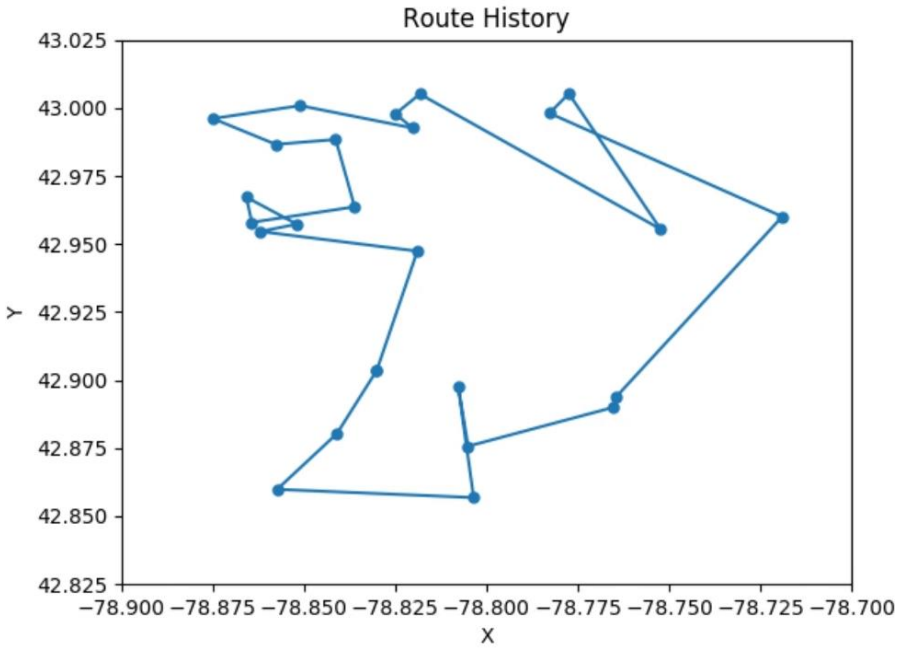


*Figure 4: Route History Animation Screenshot*

**3. Solve practice_25 Problem with GA**

Using a Windows computer running on a 3.2 Ghz 8-core AMD 8320E processor with 8gb 1600mhz ddr3 ram, the algorithm was run for ten trials. The parameters for the algorithm were set at: 800 population size, 400 iterations, .01 mutation rate, and .75 crossover rate. For each trial, the objective value (total distance) was recorded. Total run time for the test trials was ~5 hours. Results of the ten trial runs are analyzed below.

## 4. Code Validation

To validate the python implementation of the Genetic Algorithm, the results obtained in the test runs are compared to a mixed integer linear programming solution obtained with Gurobi optimizer [4]. Once installed, the Gurobi optimization software can be used in python through the gurobipy library. The MILP traveling salesman model in python for Gurobi has been previously coded in the course. The mixed-integer formulation as provided in IE555 by the Dr. Murray is shown in Fig. 5, was used for the Gurobi MILP model [5]. In the reviewed papers, "Genetic algorithms for the traveling salesman problem" by Jean-Yves Potvin and "Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation" by Gokturk Ucoluk, the genetic algorithm was shown to find robust solutions relatively close to the best known solution to the TSP problems attempted in each paper.

Table 1: Parameter notation

| | |
|---|---|
| $q$ | Number of customers. |
| $N$ | Set of all nodes in the network; $N = \{0, 1, \ldots, q\}$, where node 0 is the depot. |
| $c_{ij}$ | "Cost" of travel from node $i \in N$ to node $j \in N$, where $i \neq j$. This could be expressed as time or as distance. |

Table 2: Decision variable notation

| | |
|---|---|
| $x_{ij} \in \{0, 1\}$ | Binary decision variable that equals one if the salesperson travels from node $i \in N$ to node $j \in \{N : j \neq i\}$; $x_{ij} = 0$ otherwise. |
| $u_i \geq 0$ | Continuous decision variable used to prevent "subtours". |

The problem of minimizing the total "cost" of travel may be described via the following MILP:

$$\text{Min} \quad \sum_{i \in N} \sum_{j \in \{N: j \neq i\}} c_{ij} x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{j \in \{N: j \neq i\}} x_{ij} = 1 \quad \forall\, i \in N, \tag{2}$$

$$\sum_{i \in \{N: i \neq j\}} x_{ij} = 1 \quad \forall\, j \in N, \tag{3}$$

$$u_i - u_j + 1 \leq (q+1)(1 - x_{ij}) \quad \forall\, i \in \{1, 2, \ldots, q\}, j \in \{N : j \neq i\}, \tag{4}$$

$$x_{ij} \in \{0, 1\} \quad \forall\, i \in N, j \in \{N : j \neq i\}, \tag{5}$$

$$1 \leq u_i \leq q + 2 \quad \forall\, i \in N. \tag{6}$$

*Figure 5: MILP Traveling Salesman Formulation Excerpt [5]*

## IV. Results

As noted earlier, ten trial runs were done to analyze the performance of the genetic algorithm with PMX crossover as shown in Table 1. The best solution of the the ten trial runs had an objective value 58.02 miles. The mean objective value was 62.67 miles with a standard deviation of 3.84. Compared to results from the nearest neighbor heuristic, the GA reduced route distances by up to 16.57 miles, a significant 22% improval. In HW 4, the simulated annealing algorithm with ten test runs, average a distance of 59.73 miles with a standard deviation of 1.23. On average the simulated annealing performed 4.7% better relative to the genetic algorithm. Also,

the simulates annealing had a faster computing time at 14.95 s, relative to the 1698.94 s required for GA.

The Gurobi optimum solution is 56.444, so at it's best run the genetic algorithm is only 2.7 % worse than the Gurobi optimum solution. The results of the genetic algorithm, being very close to the optimal solution, validate this approach. As seen in the papers, the genetic algorithm with PMX crossover obtained robust solutions, very near to the optimal solution, indicating that our implementation of the GA algorithm in Python was done correctly.

From these results, it appears the genetic algorithm parameters such as crossover proability, mutation probability, and population size may not be optimized. While the solutions are robust, minimizing locally near the global minium, optimizing the parameters may leed to even better results, escaping local minimums.

*Table 1: GA Test Run Results*

| Trial # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Distance | 58.92 | 65.31 | 58.02 | 71.07 | 59.69 | 60.84 | 59.34 | 65.06 | 63.91 | 64.51 |

A unique behavior evident in the plot of average objective function value at each iteration is the oscillation that occurs in average value (Fig. 6). Although the average of the population oscillates, as seen in the plot of convergence, the best solution of each iteration does converge.

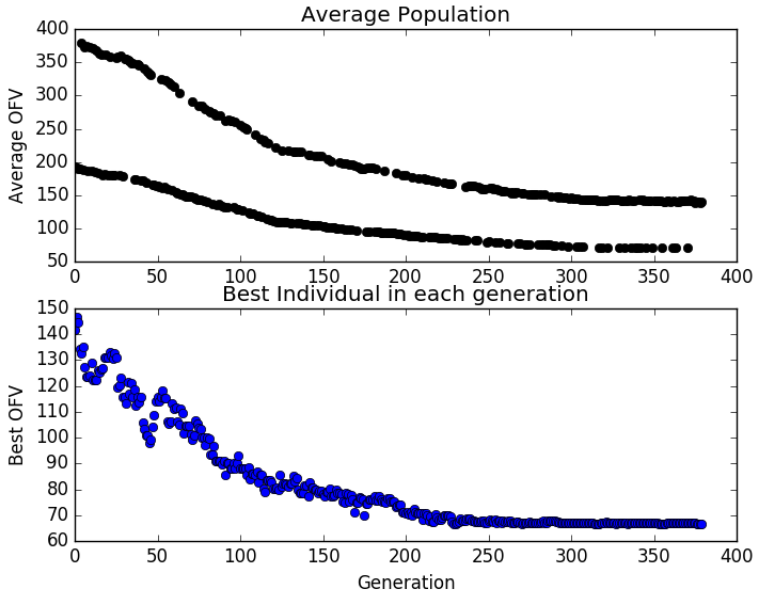


*Figure 6: Average Objective Value and Best Objective Value of Population*

A sample test run output map is shown in Fig. 6. The depot is indicated by the red marker, other locations with blue markers, and with the route shown in black lines. The output plot saved as osm.html in the problem file follows this scheme.

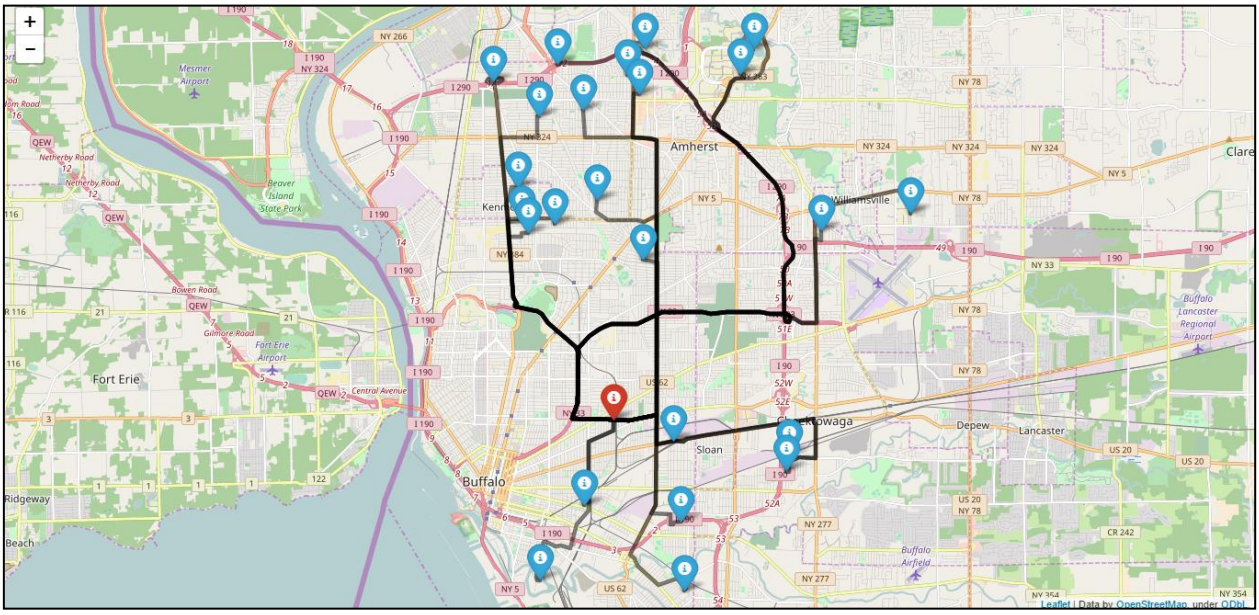


*Fig. 6: Sample Route Map*

## V. Conclusion

Testing on the genetic algorithm as it is currently set up has indicated there is room for improvement. Increasing the rate of mutation is one immediate option that may improve performace. Currently set up at .01, there is very little chance of mutation occurring, therefore route solutions may converge to local minimums and not “escape”. Further testing on population sizes may also produce better results.

Moreover, from the reviewed papers, it is evident that other crossover methodologies outperform PMX crossover, in both better routes and faster convergence. Exploring alternative crossover methods can increase the performance of the GA.

Another concern, which benefits alternative algorithms, is the computational time required for this implementation. With the current setup, for this problem, the simulated annealing approach would be recommended due to shorter computational times and better results. It is believed that with some tuning the genetic algorithm may outperform the simulated annealing in terms of getting better results, but would still take more computational time.

For concrete analysis, the GA should be implemented on more problems with a different number of cities and with different distributions. While for this problem SA outperformed GA, it is possible that in other problems the GA would outperform the SA.

## References


[1] Potvin, Jean-Yves. "Genetic algorithms for the traveling salesman problem." Annals of Operations Research 63.3 (1996): 337-370.

[2] Üçoluk, Göktürk. "Genetic algorithm solution of the TSP avoiding special crossover and mutation." Intelligent Automation & Soft Computing 8.3 (2002): 265-272.

[3] Python Software Foundation. “Python Language Reference.” Available at http://www.python.org

[4] Gurobi Optimization, Inc., "Gurobi Optimizer Reference Manual." (2016)

[5] Murray, Chase, “Mixed Integer Linear Programming (MILP) Formulation for the Traveling Salesperson Problem (TSP)” (IE555, University at Buffalo, April 7 2017)

## Appendix A

problem_25 Coordinates

| % nodeID | nodeName | isDepot | latDeg | lonDeg |
|---|---|---|---|---|
| 0 | Depot | 1 | 42.90354 | -78.8301 |
| 1 | Customer 1 | 0 | 42.8899 | -78.7655 |
| 2 | Customer 2 | 0 | 42.894 | -78.7644 |
| 3 | Customer 3 | 0 | 43.00509 | -78.8183 |
| 4 | Customer 4 | 0 | 42.99804 | -78.7828 |
| 5 | Customer 5 | 0 | 42.85985 | -78.8574 |
| 6 | Customer 6 | 0 | 42.95729 | -78.852 |
| 7 | Customer 7 | 0 | 42.99607 | -78.875 |
| 8 | Customer 8 | 0 | 42.95545 | -78.7523 |
| 9 | Customer 9 | 0 | 42.94738 | -78.819 |
| 10 | Customer 10 | 0 | 42.85684 | -78.8035 |
| 11 | Customer 11 | 0 | 42.96007 | -78.7189 |
| 12 | Customer 12 | 0 | 42.8757 | -78.8051 |
| 13 | Customer 13 | 0 | 42.96365 | -78.8362 |
| 14 | Customer 14 | 0 | 42.96719 | -78.8659 |
| 15 | Customer 15 | 0 | 42.95795 | -78.8643 |
| 16 | Customer 16 | 0 | 42.99263 | -78.8203 |
| 17 | Customer 17 | 0 | 43.00083 | -78.8511 |
| 18 | Customer 18 | 0 | 43.00524 | -78.7774 |
| 19 | Customer 19 | 0 | 42.9866 | -78.8578 |
| 20 | Customer 20 | 0 | 42.89774 | -78.8078 |
| 21 | Customer 21 | 0 | 42.95458 | -78.862 |
| 22 | Customer 22 | 0 | 42.99779 | -78.825 |
| 23 | Customer 23 | 0 | 42.98839 | -78.8413 |
| 24 | Customer 24 | 0 | 42.88031 | -78.841 |