# IE 555 – Programming for Analytics

# Simulated Annealing Report

Name: Arka Chowdhury

UB#50206726

## Flow of the code

The flow of the code is represented in Figure 1: Flowchart of the travelling salesman problem. Initially we import all the libraries that would be required for our code to function. We then initialize the values of initial temperature, iteration, delta value, final temperature and cut off time. The inputs like locations folder, objective value, nearest neighbor, integer programming and simulated annealing selection type, type of map plot are captured from the command lines. Using the locations folder we import the values in the csv folder of Problem_3.csv or Problem_25.csv. We create a node using the class 'make_node' and store all the details about the nodes. We use the function genTravelMatrix to get the distance and time data frame among the nodes from Mapquest using valid mapquest key. Now we solve the TSP problem and depending on our selection for the algorithm/s we have chosen to get the output. The selection goes to the respective chosen function and gives the tour and its cost.

The flow in Simulate Annealing function is described in Figure 2: Flowchart of the simulated annealing. In this function we receive all the required input like the data frames and the initialized variables required for simulated annealing. Initialize the start time and all the lists which would eventually store the points which has bad values and good values to plot. Call the nearest neighbor function to get its solution. Then we start with the Phase I of simulated annealing algorithm where we set the current tour and best tour as the nearest neighbor tour. Phase II starts with the iteration where we append in a list the Z values. 5 subtours reversals are generated using a while loop. We generate the subtour reversal by randomly selecting the starting and the end point and then reversing the selected part to get the tour. Depending on the objective type the cost is calculated and checked if it's the minimum cost among the 5. Then we check if the subtour cost is less than the current cost, if yes, we change the current tour to the reversed subtour, if no, then we check that if the reversed subtour is acceptable. The good solution, the bad solution accepted and the bad solution rejected is appended in lists for further plotting. We check if the current solution is better than best solution, then we change the best solution with the current. We finally check the breaking condition that if we have reached the final temperature or reached the cut-off time. We finally plot all the points and generate a plot to compare the cost value along with time.

After finding all the solutions for the respective algorithms we use Folium to plot the tours. We first check the depots and mark it in red. According to the given input we select if we need to plot the map turn by turn lines or with polylines. We use the cost values in each optimization technique to determine its presence and hence plot the tours. Finally we print the tours and their respective costs as output.

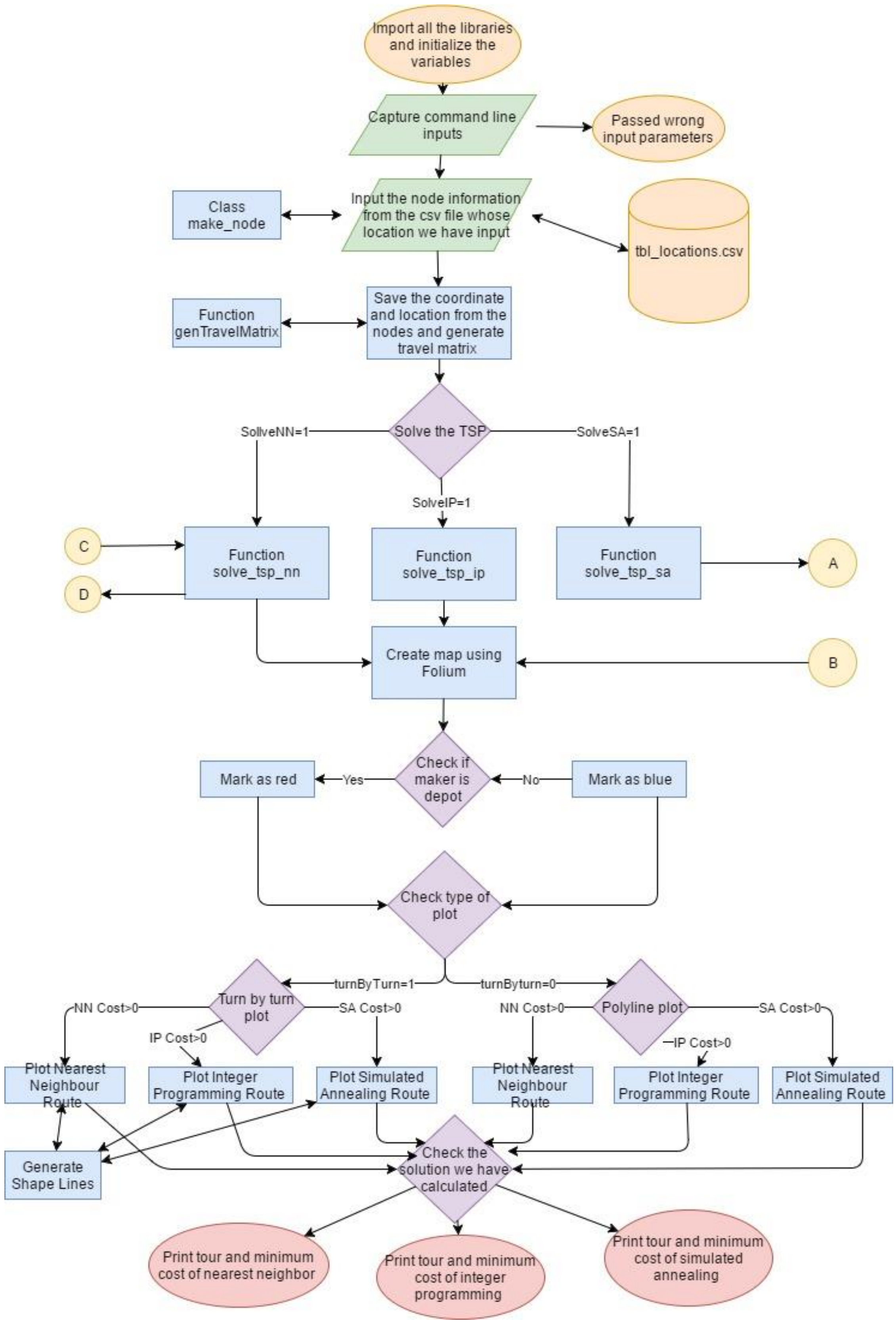


Figure 1: Flowchart of the Traveling Salesman Problem

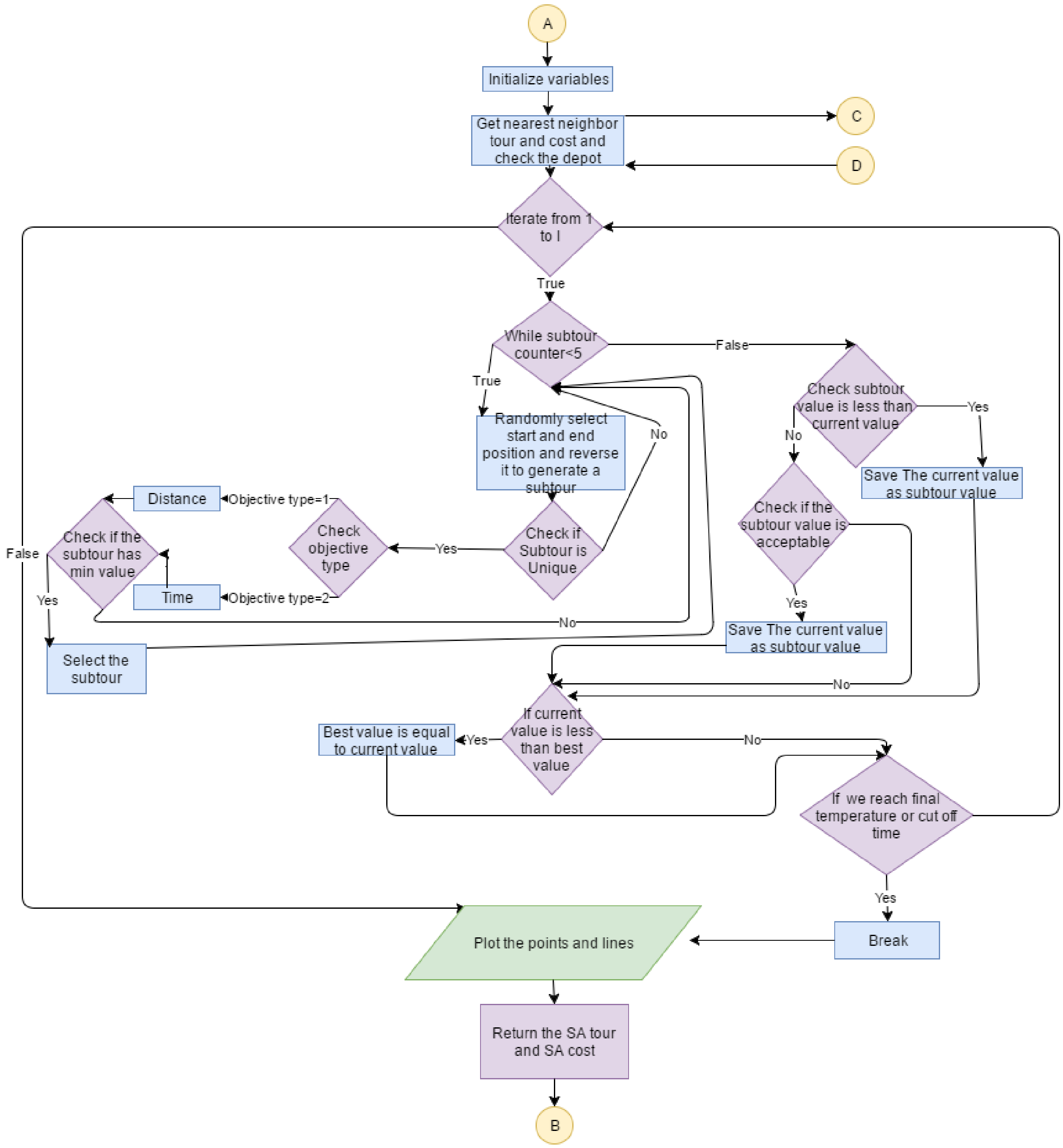


Figure 2: Flowchart of the Simulated Annealing

## Simulated Annealing Heuristic Tuning

For tuning the Simulated Annealing 3 temperatures, 6 delta values and 5 Iteration values was selected. We calculated the costs for each combinations so that we could get an essence of which combination is giving us the best value. T=200, 500 and 1000. Delta= 0.5, 0.1, 0.05, 0.01, 0.005 and 0.001. I= 500, 1000, 5000, 10000 and 50000. The Final temperature was 0 for all combinations. We selected distance between the nodes as objective value for the following calculations.

The cost function for the distance table is as follows

**For Initial Temperature=200**

| Delta\Iteration | 500 | 1000 | 5000 | 10000 | 50000 |
|---|---|---|---|---|---|
| **0.5** | 70.503 | 65.929 | 67.456 | 69.392 | 66.443 |
| **0.1** | 72.939 | 66.655 | 68.613 | 37.825 | 65.938 |
| **0.05** | 71.134 | 69.363 | 66.824 | 68.36 | 69.647 |
| **0.01** | 68.41 | 70.33 | 70.33 | 68.594 | 60.473 |
| **0.005** | 69.393 | 67.456 | 71.061 | 62.942 | 61.252 |
| **0.001** | 67.84 | 70.538 | 65.966 | 67.253 | 65.614 |

Table 1: Initial Temperature 200

**For Initial Temperature=500**

| Delta\Iteration | 500 | 1000 | 5000 | 10000 | 50000 |
|---|---|---|---|---|---|
| **0.5** | 70.33 | 69.509 | 69.509 | 74.591 | 70.443 |
| **0.1** | 69.603 | 71.576 | 71.576 | 67.115 | 67.413 |
| **0.05** | 72.576 | 70.33 | 70.33 | 65.693 | 67.29 |
| **0.01** | 69.363 | 70.94 | 70.94 | 69.809 | 64.406 |
| **0.005** | 64.298 | 67.98 | 67.98 | 70.069 | 65.845 |
| **0.001** | 71.852 | 70.475 | 70.475 | 67.571 | 64.629 |

Table 2: Initial Temperature 500

**For Initial Temperature=1000**

| Delta\Iteration | 500 | 1000 | 5000 | 10000 | 50000 |
|---|---|---|---|---|---|
| **0.5** | 70.055 | 67.571 | 72.945 | 69.029 | 71.966 |
| **0.1** | 71.218 | 68.502 | 67.736 | 64.983 | 69.921 |
| **0.05** | 71.64 | 69.393 | 73.74 | 69.208 | 69.481 |
| **0.01** | 74.591 | 73.861 | 68.964 | 67.838 | 64.843 |
| **0.005** | 69.754 | 69.454 | 70.333 | 66.461 | 65.791 |
| **0.001** | 67.429 | 73.773 | 71.143 | 69.032 | 67.633 |

Table 3: Initial Temperature 1000

**From the above observations we found the optimum value of 60.473 at Initial Temperature=200, Final temperature=0, delta=0.01 and iteration=50,000.**

The cost function for the time table is as follows

**For Initial Temperature=200**

| Delta\Iteration | 500 | 1000 | 5000 | 10000 | 50000 |
|---|---|---|---|---|---|
| **0.5** | 8327 | 8327 | 8327 | 8248 | 8327 |
| **0.1** | 8327 | 8327 | 8165 | 8314 | 8327 |
| **0.05** | 8136 | 8248 | 8178 | 8322 | 8327 |
| **0.01** | 8327 | 8327 | 8050 | 8143 | 8054 |
| **0.005** | 8327 | 8327 | 8327 | 8089 | 8054 |
| **0.001** | 8274 | 8327 | 8327 | 8054 | 8140 |

Table 4: Initial Temperature 200

**For Initial Temperature=500**

| Delta\Iteration | 500 | 1000 | 5000 | 10000 | 50000 |
|---|---|---|---|---|---|
| **0.5** | 8327 | 8327 | 8327 | 8327 | 8323 |
| **0.1** | 8327 | 8327 | 8327 | 8193 | 8137 |
| **0.05** | 8327 | 8327 | 8327 | 8327 | 8179 |
| **0.01** | 8327 | 8327 | 8327 | 8327 | 8047 |
| **0.005** | 8327 | 8327 | 8327 | 8327 | 8226 |
| **0.001** | 8327 | 8327 | 8327 | 8327 | 8327 |

Table 5: Initial Temperature 500

**For Initial Temperature=1000**

| Delta\Iteration | 500 | 1000 | 5000 | 10000 | 50000 |
|---|---|---|---|---|---|
| **0.5** | 8327 | 8327 | 8327 | 8327 | 8327 |
| **0.1** | 8327 | 8327 | 8327 | 8327 | 8321 |
| **0.05** | 8327 | 8327 | 8327 | 8327 | 8080 |
| **0.01** | 8327 | 8327 | 8327 | 8327 | 8327 |
| **0.005** | 8327 | 8327 | 8327 | 8327 | 8327 |
| **0.001** | 8327 | 8327 | 8327 | 8327 | 8327 |

Table 6: Initial Temperature 1000

**From the above observations we found the optimum value of 8054 at Initial Temperature=200, Final temperature=0, delta=0.001/0.005 and iteration=10,000/50,000 respectively.**

From the above observations we found that more the iterations, we have greater chances of finding a good value. For delta we can say that it has an optimum value around the value 0.05 and 0.01. We see that in lesser values and greater values of delta, the cost value is increasing. Delta also determines the number of iterations. We can also observe in Table 1 that T=200 is giving more better solution than the other two in case of distance. By seeing the values in the temperature we can say that there is a possibility to get better distance values if we decrease the temperature . In case of objective value of temperature we can see from Table 4, Table 5 and Table 6, that higher temperature and lower iterations are not giving a better solution. Some particular combinations, which would work poorly, will be having low temperature and high delta. This would not allow much iterations to find the good solutions as temperature will decrease very soon.

## Running the heuristic for 10 times

| Sl No | Iteration | Objective Value | Runtime(seconds) |
|---|---|---|---|
| 1 | 50000 | 67.099 | 35.1769998074 |
| 2 | 50000 | 63.609 | 35.2909998894 |
| 3 | 50000 | 67.12 | 35.2550001144 |
| 4 | 50000 | 59.431 | 34.9769999981 |
| 5 | 50000 | 59.81 | 35.2829999924 |
| 6 | 50000 | 64.746 | 35.4390001297 |
| 7 | 50000 | 66.89 | 34.8800001144 |
| 8 | 50000 | 62.728 | 34.9659998417 |

| | | | |
|---|---|---|---|
| 9 | 50000 | 65.623 | 34.9679999352 |
| 10 | 50000 | 59.71 | 35.2149999142 |

Table 4: Observation table for 10 runs

We don't get the same solution each time because each time we run the program the subtour reversals are generated randomly. The random subtour reversals have different cost values at different time of the iteration causing selection of different current solution, which would ultimately lead to the best solution. It also randomly accepts values whose probability decrease with the decrease in the best solution. In the above table we can see that the best solution ranges from 59.431 to 67.12. The runtime also changes which depends on the time taken to iterate through the algorithm. Thus we can conclude that as every time the code is run we randomly generate subtours and randomly accept current solution which are not good, hence we randomly generate outputs in each run.

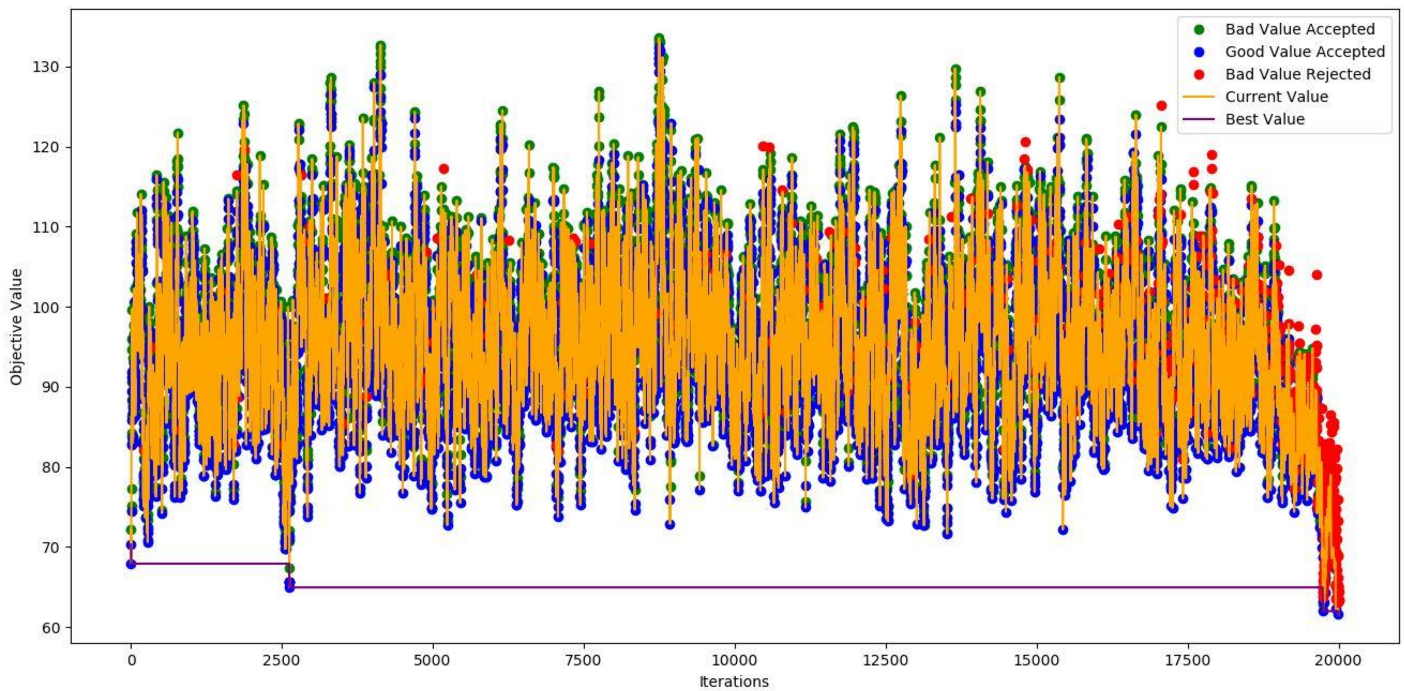


Figure 3: Simulated Annealing heuristic over time

We can refer to Figure 3: Simulated Annealing heuristic over time and infer some interesting observations.

Figure 3 depicts the flow of the objective value of the heuristics along with the iterations. In the figure the there are two lines: the line in orange is the current value while the line in purple is the best value. There are 3 dot markers to represent 3 objective values: The blue dot shows the good values which were accepted, the green dot shows the bad values which were accepted and the red dot shows the bad values which were rejected. Thus in the figure we can observe how with increase in iterations the we are getting the best value in the last. In the figure we can also see that as the iteration is increasing we are rejecting bad values more than we are accepting the bad ones.

As the green dots indicate the bad value accepted we see that early in the process there are many bad values that are accepted as compared to the bad values that are rejected(red dots). As the temperature cools we can observe that the red dots increase significantly and hardly we can see any green dots when the temperature reaches the final temperature. We can also observe the heuristic is converging to a locally optimal solution as the current value is quite less near the final temperature.

### Comparison between the solution of Gurobi and Mixed Integer Linear Programming

We executed the code using the command "python solve_tsp_3.py practice_25 1 0 1 1 120 1" in the console to run the MILP and SA heuristics.

```
Runtime of SA: 34 seconds
```

Figure 4: SA Runtime

```
Explored 1449 nodes (13376 simplex iterations) in 1.51 seconds
```

Figure 5: MILP(Gurobi) Runtime

Comparing Figure:4 and Figure:5 we can observe that using gurobi the runtime is a way more less than the runtime used in Simulate Annealing. The runtime of Gurobi is about 4.41% of the time taken by Simulated Annealing to reach its optimal value.

```
IP Route:
[0, 24, 5, 10, 12, 1, 2, 8, 11, 4, 18, 22, 16, 3, 17, 7, 19, 23, 13, 6, 15, 14, 21, 9, 20, 0]
IP 'cost':
56.444

SA Route:
[0, 24, 5, 10, 12, 1, 2, 20, 9, 7, 17, 3, 16, 22, 18, 4, 8, 11, 13, 23, 19, 14, 15, 6, 21, 0]
SA 'cost':
66.076
```

Figure 6: Route and objective values for distance

Figure 6 shows that the objective function value for MILP is much less than the cost of SA.

The runtime and the objective values clearly shows that Mixed Integer Linear Programming is a better choice to find the optimal solution than Simulated Annealing. The cost for MILP is about 15% less than the cost we get by Simulated Annealing.

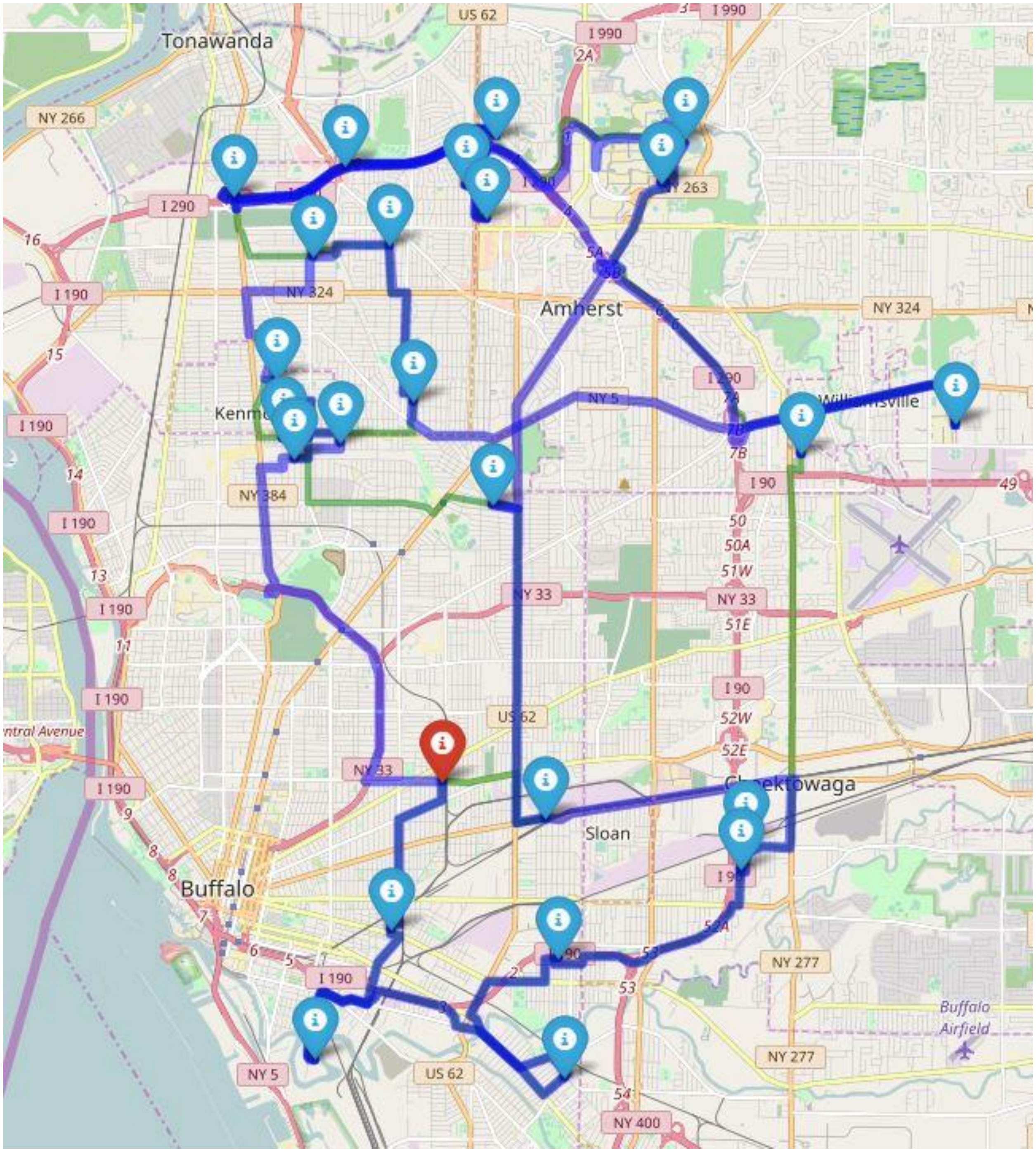


Figure 7: Map showing the route of SA in blue and MILP in green.

**Instructions for running the code**

- The file name of the code is ARKACHOW_code.py
- Make sure that the folder in which the file is present also has a folder named Problems which further has a folder named practice_25 which contains a file named tbl_locations.csv, it contains the location information.
- Open your terminal and go to the folder location where the file is present.
- Make sure you have the libraries random, matplotlib, numpy, time and math as additional libraries to run this code.
- Run the code along with the command line inputs.
- The input to be given in the command line while executing the code is given below
  - Locations Folder: practice_25 (practice 3 will not work to generate 5 unique subtours)
  - Objective Type: 1- to minimize distance, 2- to minimize time
  - 1- to solve using nearest neighbor, 0- to not solve using nearest neighbor
  - 1- to solve using mixed integer linear programming, 0- to not solve using nearest neighbor
  - 1- to solve using simulated annealing, 0- to not solve using simulated annealing
  - Cut off time: 1- for no cut off time, ‘value’ in seconds for Gurobi cut off time in IP.
  - Map Route Display Type: 1- For detailed route, 0- Draw straight lines joining the points
- Sample line to be written in the terminal to run only simulated annealing- python solve_tsp_3.py practice_25 1 0 0 1 120 1
- Code to run to compare MILP and SA- python solve_tsp_3.py practice_25 1 0 1 1 120 1
- Code to run to get the optimum time from all the heuristics- python solve_tsp_3.py practice_25 2 1 1 1 120 1