Traveling Salesman Problem – Closest Edge Insertion Heuristic (Greedy 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 the Closest-Edge Insertion Heuristic approach using a variant of the greedy 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 Brute Force and Breadth-First Search (BFS) & Depth-First Search (DFS) approaches that were explored in the last two projects to analyze differences between the algorithm runtimes when dealing with complex TSP datasets where approaches like Brute Force, BFS, or DFS would be inefficient as the number of cities increase, as seen in the last couple 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 closest-edge insertion heuristic, there were two consecutive steps that were taken within the modified greedy algorithm: 1) finding a node (city) that is closest to one of the existing edges and 2) breaking the closest existing edge to make two new edges to the closest node. To simplify the vocabulary used in this report, cities within the TSP file will be referred to as 'nodes' and line segments (paths) that connect two cities together at a time are defined by 'edges.' All calculations were based and dependent on a triangular structure. To begin, a starting node is assigned to the startCity variable. Next, the closest node to that startCity is found through the function startingPair which uses the basic distance formula and iterates over every city within the coordinates list to find the closest city. Then, an edge is formed between these two cities and appended to the list visitedEdges. Now that there exists an edge within the whole graph, one can proceed to finding the closest node to the first edge that has been created above and break off that existing edge to lead to that specific node as outlined in the two consecutive steps briefly described above. A more elaborate explanation on the technique in performing the two steps can be found under Section 2.1 and 2.2 respectively. The following visual representation shows the above technique:

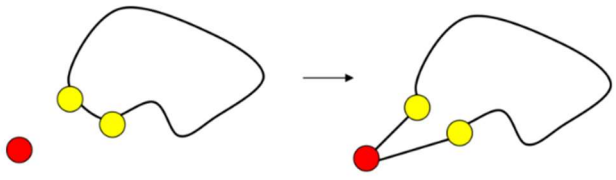
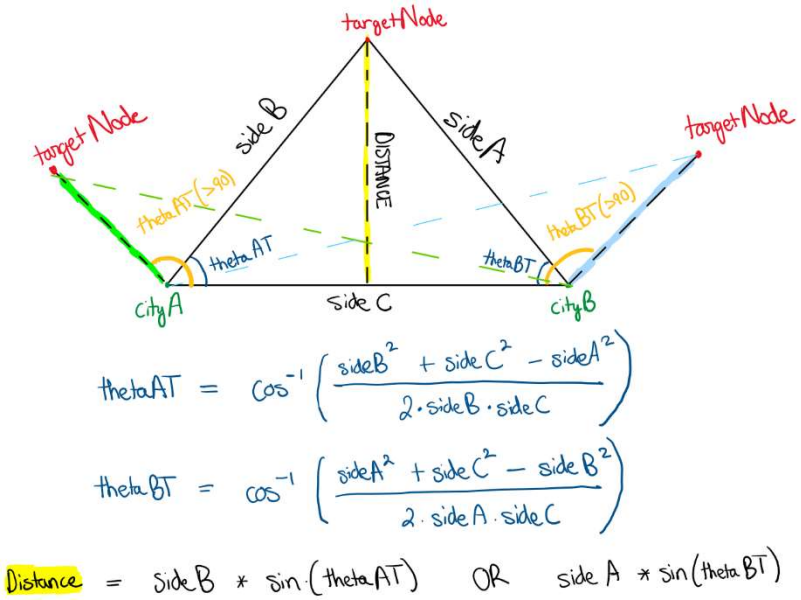


Figure from: www.cs.uu.nl/docs/vakken/an/an-tsp.ppt

2.1. Finding the Closest Node given an existing Edge

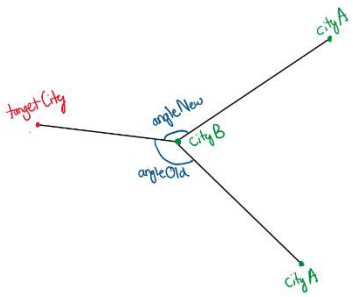
In order to find the node that is closest to a given list of existing edges to make the next progression, there is 1 required step and 3 conditionals based on the output of the said first step. The first required step is implementing the Law of Cosines to find the angles that span out to the current target city (thetaAT, thetaBT) because we can calculate the three sides of the potential triangle using the coordinates and the distance formula. If the thetaAT and thetaBT that were found above are both acute (less than 90 degrees), then one can proceed to use the basic geometry concept of SOH CAH TOA and the perpendicular property of the opposite line to the edge to apply the trigonometrical function sin to find the opposite segment's length (the node's perpendicular distance from the edge) since we know the length of the hypotenuse, which would just be distance between one of the cities on the existing edge to the target city. This is done through the nodeDistance function. If thetaAT is greater than 90 degrees (obtuse), then the distance between the edge and the target node is simply the distance between the target city and the node (city on edge) where the obtuse angle occurs (cityA) which can be computed using the simple distance formula. Likewise, if thetaBT is greater than 90 degrees (obtuse), then the distance between the edge and the target node is simply the distance between the target city and cityB. The following figure visually represents the process described above:



2.2 Breaking the existing Edge to connect to the new Closest Node

After the above method repetitively checks each unvisited city in the cities list for every existing edge in the visitedEdges list in a nested for loop structure, the distance is recorded every iteration and is compared against the shortestDistance by far (assigned to the first iteration's distance if shortestDist == 0) to pick the correct edge to break. If the current iteration's distance is shorter than the shortestDistance recorded by far, the variables of the best edge (with the two cities), the best target city, and the shortestDistance are updated. In the special case where the current iteration's distance is equivalent to the shortestDistance by far, the angleFromEdge function is called to break the tie by comparing the angles at the

pivot city for each of the equivalent edges (thetaAT or thetaBT for each) as shown to the right in the hand drawn figure:



From the figure, it can be seen how angleNew is lesser than angleOld. In this case, the algorithm would pick the upper edge and break it to form the next two edges.

After the closest edge to the target city and the perfect edge with the two cities to break have been found, the algorithm then removes the edge that breaks up from the visitedEdges list since it does not exist anymore from the partition done above. Additionally, it removes the target node found from the list of unvisited cities. Finally, it appends the two new edges that have been formed by breaking the existing edge to the visitedEdges list. This process occurs repetitively until all of the cities have been visited and added on to the visitedEdges list in some form.

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

The closest-edge insertion heuristic was significantly more efficient than any of the approaches, Brute Force and BFS/DFS, taken in the last couple projects. While just a 12-city dataset took over 2 hours using a brute force approach, a 40-city dataset took about 0.0867 seconds which is a significant increase in efficiency and shows how the modified greedy algorithm performed really well as seen below in the outputs and figures.

3.1 Data

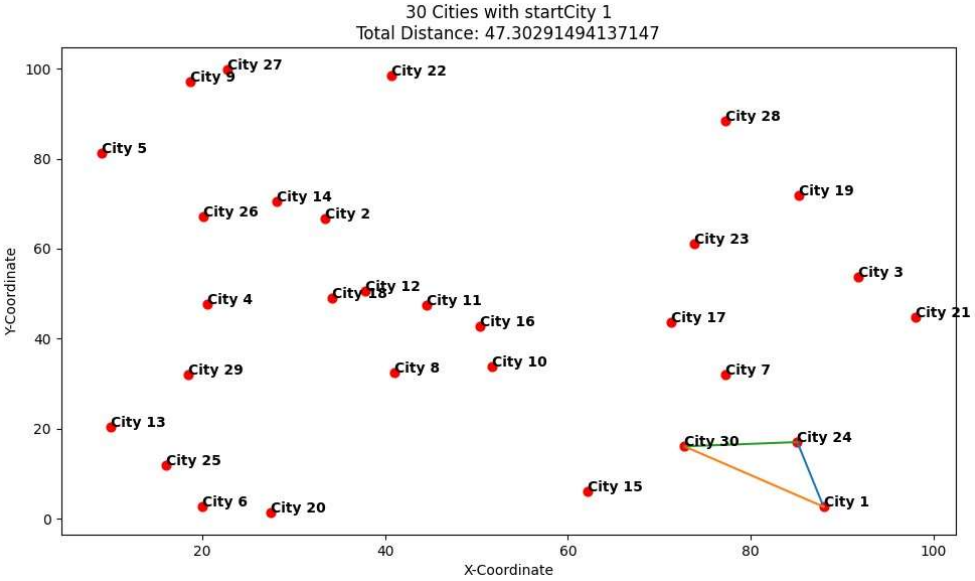
The data being inputted was provided in the form of a .tsp file which had included a list of cities and their corresponding x and y coordinates. More specifically, there were two datasets being used that included all coordinates for 30 and 40 different cities respectively. Each file was split into a 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 (30 Cities):

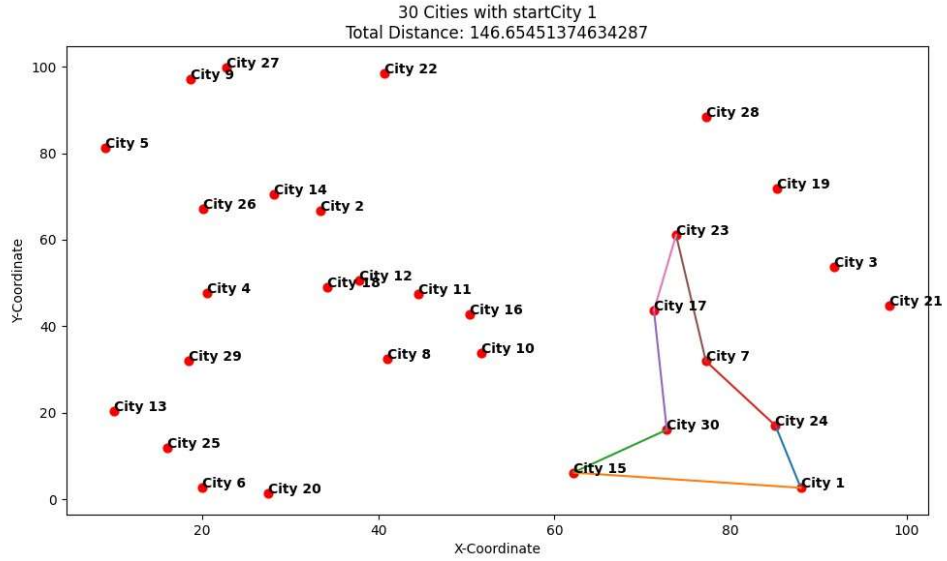
```
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
5 9.006012 81.185339
6 20.032350 2.761925
7 77.181310 31.922361
8 41.059603 32.578509
9 18.692587 97.015290
10 51.658681 33.808405
11 44.563128 47.541734
```

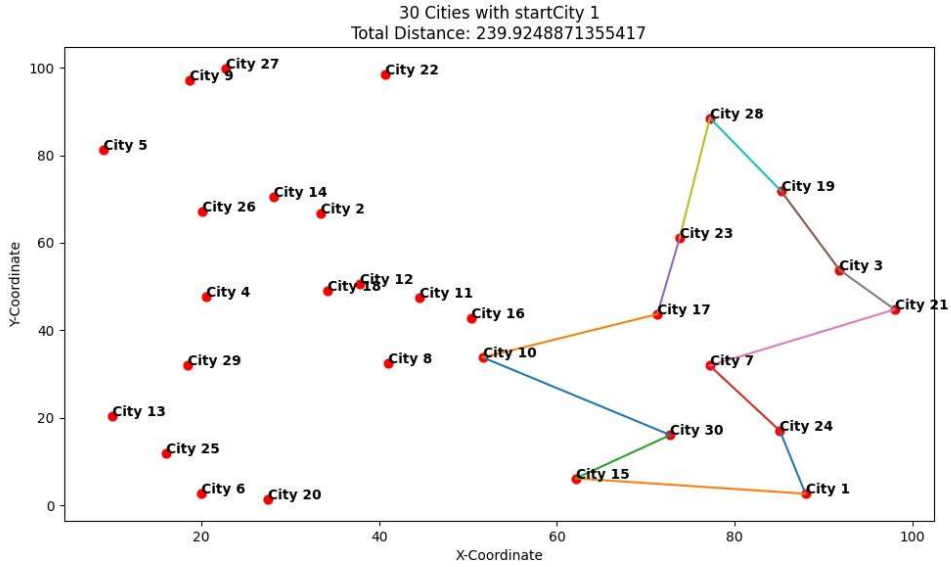
3.2 Results

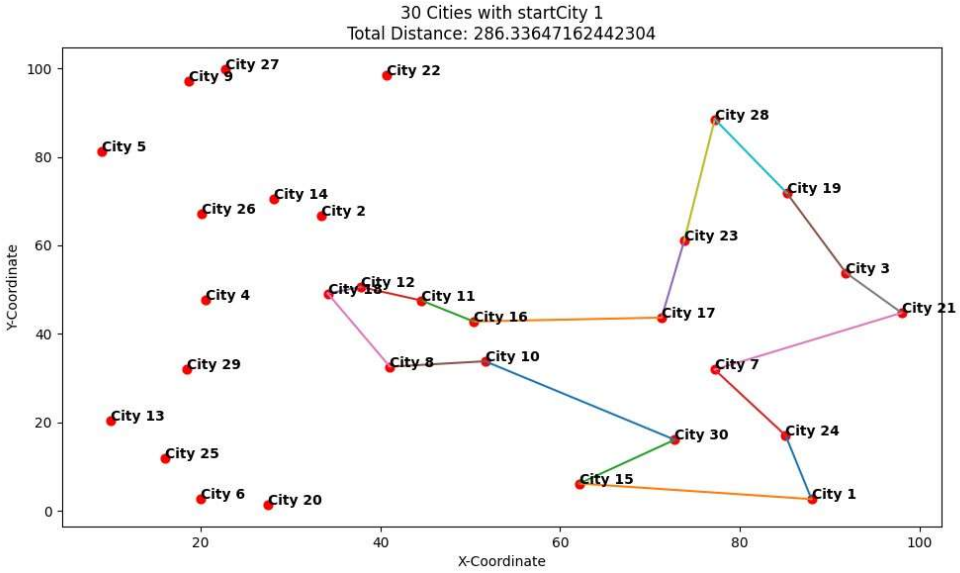
```
#### 30 CITIES (startCity 1) ####
Order of Nodes Inserted: [1, 24, 30, 15, 7, 17, 23, 19, 3, 21, 28, 10, 16, 11, 12, 18, 8, 4, 29, 13, 25, 6, 20, 2, 14, 26, 5, 9, 27, 22]
Closest Edge Insertion Heuristic Runtime: 0.03692439999999997
Total Cost for 30 Cities: 506.3786001401126
```

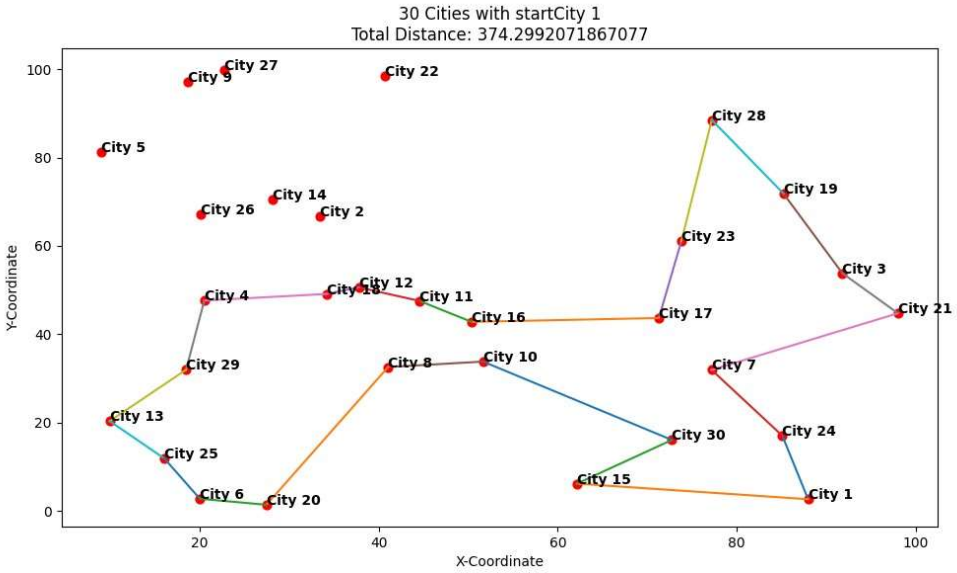
```
#### 40 CITIES (startCity 7) ####
Order of Nodes Inserted: [7, 17, 40, 10, 16, 11, 12, 18, 8, 37, 35, 4, 33, 38, 29, 31, 23, 34, 13, 25, 6, 20, 26, 14, 2, 19, 30, 24, 15, 1, 36, 28, 32, 39, 3, 21, 5, 9, 27, 22]
Closest Edge Insertion Heuristic Runtime: 0.08673339999999996
Total Cost for 40 Cities: 615.781573659662
```

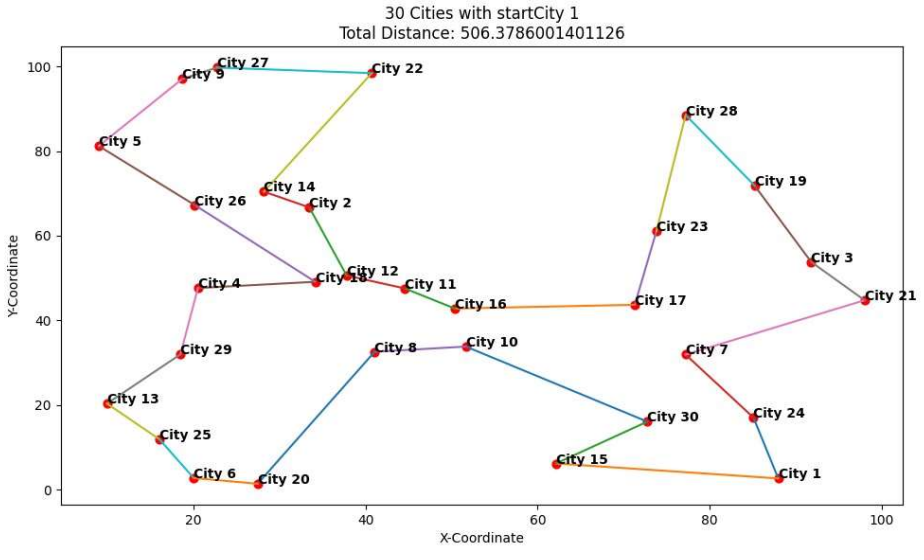

30 Cities with startCity 1
Total Distance: 47.30291494137147
City 27
City 9
City 22
City 28
City 5
City 19
City 14
City 26
City 2
City 23
City 3
City 12
City 18
City 4
City 11
City 21
City 16
City 17
City 29
City 8
City 10
City 7
City 13
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate

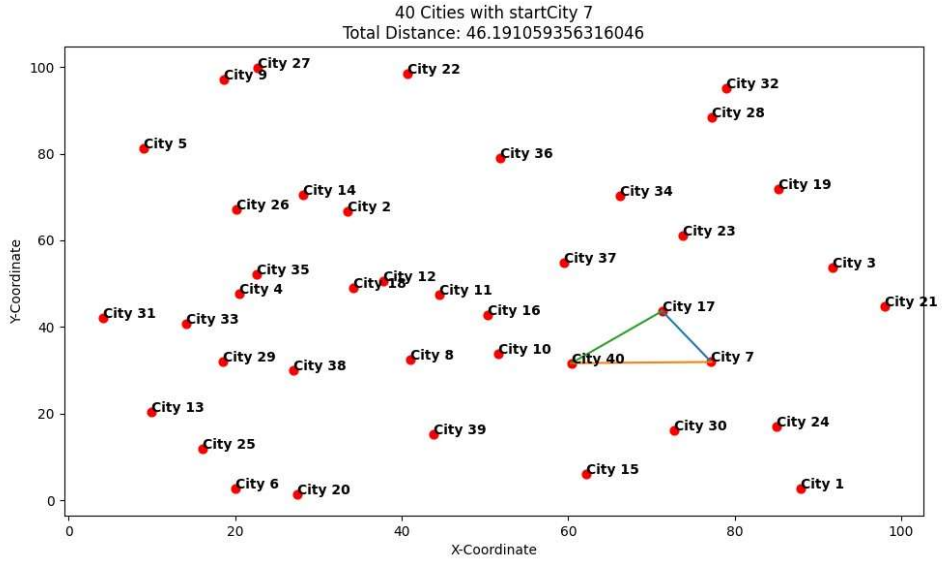

30 Cities with startCity 1
Total Distance: 146.65451374634287
City 27
City 9
City 22
City 28
City 5
City 19
City 14
City 26
City 2
City 23
City 3
City 12
City 18
City 4
City 11
City 21
City 16
City 17
City 29
City 8
City 10
City 7
City 13
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate

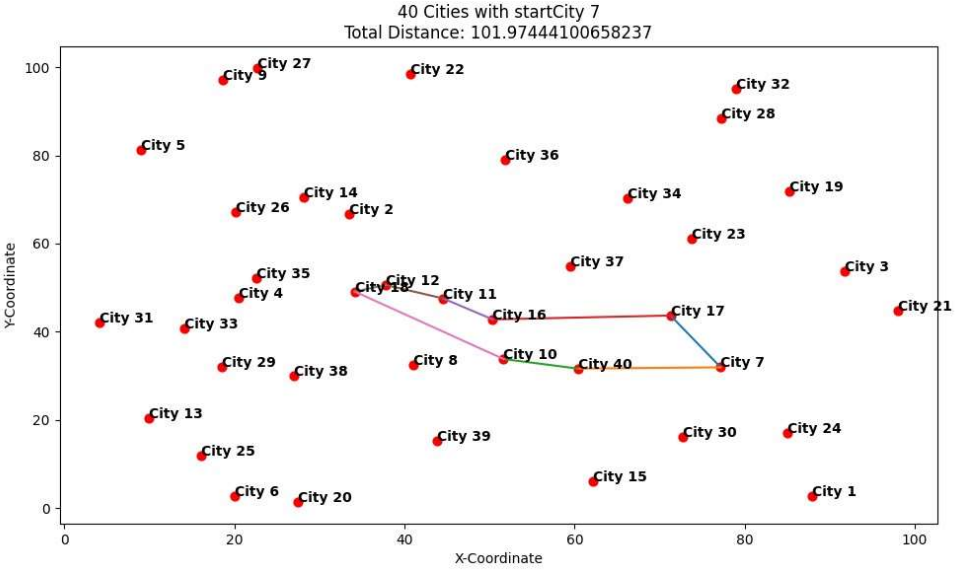

30 Cities with startCity 1
Total Distance: 239.9248871355417
City 27
City 9
City 22
City 28
City 5
City 19
City 14
City 26
City 2
City 23
City 3
City 12
City 18
City 4
City 11
City 21
City 16
City 17
City 29
City 8
City 10
City 7
City 13
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate

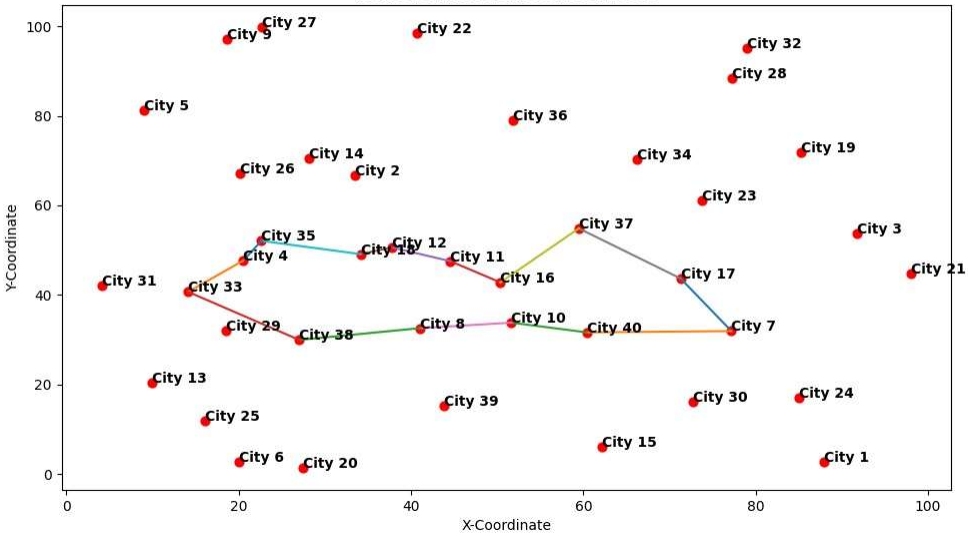

30 Cities with startCity 1
Total Distance: 286.33647162442304
City 27
City 9
City 22
City 28
City 5
City 19
City 14
City 26
City 2
City 23
City 3
City 4
City 12
City 11
City 21
City 16
City 17
City 29
City 8
City 10
City 7
City 13
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate

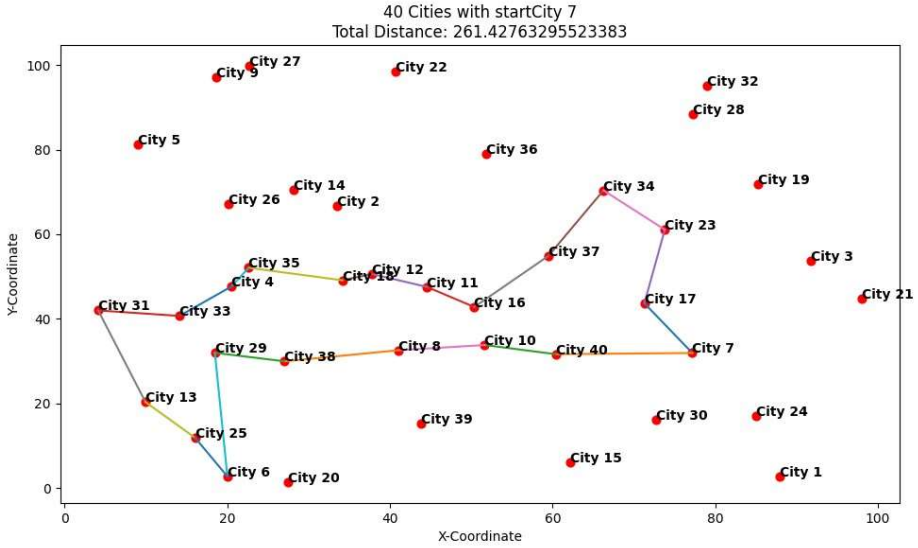

30 Cities with startCity 1
Total Distance: 374.2992071867077
City 27
City 9
City 22
City 28
City 5
City 19
City 14
City 26
City 2
City 23
City 3
City 4
City 12
City 11
City 21
City 16
City 17
City 29
City 8
City 10
City 7
City 13
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate

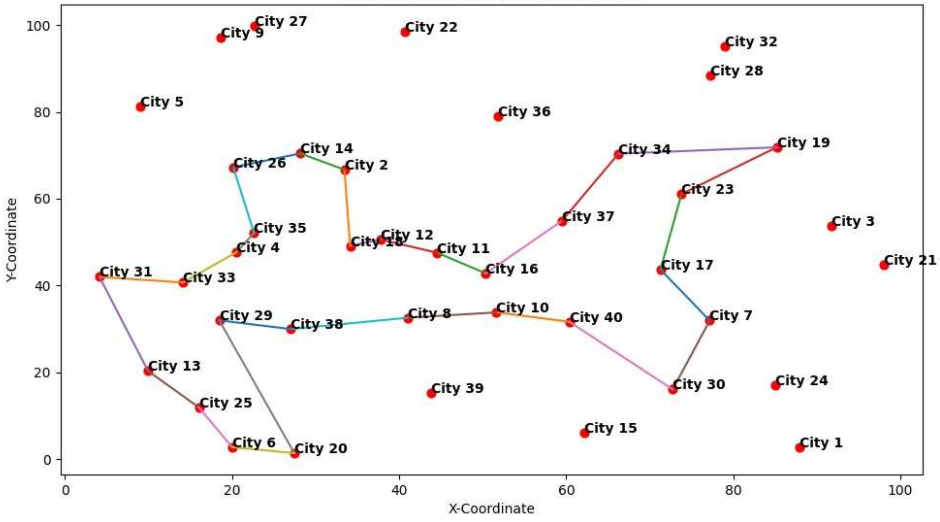

30 Cities with startCity 1
Total Distance: 506.3786001401126
City 27
City 9
City 22
City 28
City 5
City 19
City 14
City 26
City 2
City 23
City 3
City 4
City 12
City 11
City 21
City 16
City 17
City 29
City 8
City 10
City 7
City 13
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate

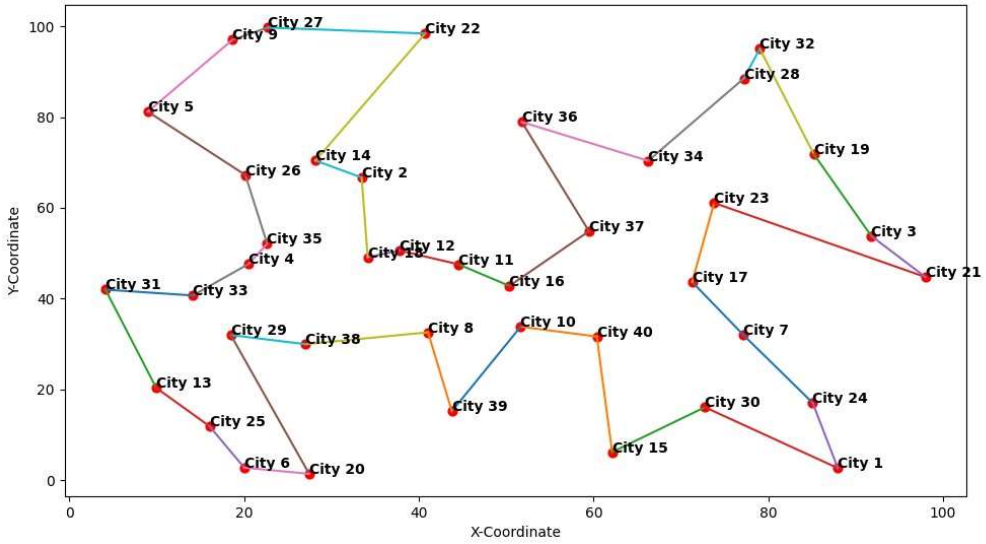

40 Cities with startCity 7
Total Distance: 46.191059356316046
City 27
City 9
City 22
City 32
City 28
City 5
City 36
City 14
City 34
City 19
City 26
City 2
City 23
City 37
City 35
City 3
City 12
City 4
City 11
City 31
City 16
City 17
City 21
City 33
City 29
City 38
City 8
City 10
City 40
City 7
City 13
City 39
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate


40 Cities with startCity 7
Total Distance: 101.97444100658237
City 27
City 9
City 22
City 32
City 28
City 5
City 36
City 14
City 34
City 19
City 26
City 2
City 23
City 37
City 35
City 3
City 12
City 4
City 11
City 31
City 16
City 17
City 21
City 33
City 29
City 38
City 8
City 10
City 40
City 7
City 13
City 39
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate


40 Cities with startCity 7
Total Distance: 157.1057783416503
City 27
City 9
City 22
City 32
City 28
City 5
City 36
City 14
City 34
City 19
City 26
City 2
City 23
City 37
City 35
City 3
City 12
City 4
City 11
City 31
City 16
City 17
City 21
City 33
City 29
City 38
City 8
City 10
City 40
City 7
City 13
City 39
City 30
City 24
City 25
City 15
City 6
City 20
City 1
Y-Coordinate
X-Coordinate


40 Cities with startCity 7
Total Distance: 261.42763295523383
City 27
City 9
City 22
City 32
City 28
City 5
City 36
City 14
City 34
City 19
City 26
City 2
City 23
City 37
City 3
City 35
City 12
City 4
City 11
City 21
City 31
City 33
City 16
City 17
City 29
City 38
City 8
City 10
City 40
City 7
City 13
City 39
City 30
City 24
City 25
City 15
City 6
City 20
City 1
X-Coordinate
Y-Coordinate


40 Cities with startCity 7
Total Distance: 350.03295565723545
City 27
City 9
City 22
City 32
City 28
City 5
City 36
City 14
City 34
City 19
City 26
City 2
City 23
City 37
City 3
City 35
City 12
City 4
City 11
City 21
City 31
City 33
City 16
City 17
City 29
City 38
City 8
City 10
City 40
City 7
City 13
City 39
City 30
City 24
City 25
City 15
City 6
City 20
City 1
X-Coordinate
Y-Coordinate


40 Cities with startCity 7
Total Distance: 615.781573659662
City 27
City 9
City 22
City 32
City 28
City 5
City 36
City 14
City 34
City 19
City 26
City 2
City 23
City 37
City 3
City 35
City 12
City 4
City 11
City 21
City 31
City 33
City 16
City 17
City 29
City 38
City 8
City 10
City 40
City 7
City 13
City 39
City 30
City 24
City 25
City 15
City 6
City 20
City 1
X-Coordinate
Y-Coordinate

4. Discussion

As seen above, the figures and the outputs for the two datasets (30-city and 40-city) show very efficient results with absolutely no crossovers in every progression for each dataset which is a great sign verifying the closest-edge insertion heuristic, with edge cases, that we have implemented. Even when dealing with datasets as big as 40 cities, this approach proved to be a lot more efficient than the approaches that were previously used. This additionally proves that the greedy algorithm variation for the closest-edge insertion heuristic works much better when the number of cities increase in datasets.

Although the greedy algorithm implemented above for the closest-edge insertion heuristic works perfectly with no crossovers and a very efficient time when compared to other approaches, it does come with a disadvantage. While the Brute Force approach takes a few hours to compute a solution, it is guaranteed to find the optimal solution. However, in the case of the greedy algorithm or BFS/DFS, it's not always guaranteed to find the most optimal solution with the least path. The greedy algorithm in specific only makes an estimate to what could be the optimal solution. Nevertheless, the advantageous efficiency and the time complexity of the greedy algorithm when it comes to larger datasets can be very beneficial when one attempts to cut down computing power or aim for quick results.

5. References

Matplotlib Documentation - https://matplotlib.org/stable/index.html
Closest Insertion Heuristic Figure - www.cs.uu.nl/docs/vakken/an/an-tsp.ppt
Law of Cosines Explanation - https://www.mathsisfun.com/algebra/trig-solving-sss-triangles.html