**Project 3: TSP - Closest Edge Insertion Heuristic**

Stone Barrett

[Computer Science Engineering]

Speed School of Engineering

University of Louisville, USA

[scbarr04@louisville.edu](mailto:scbarr04@louisville.edu)

1. **Introduction**

In this project I was asked to implement a variation of what is traditionally called the greedy approach to solving the Traveling Salesperson Problem. This project’s downloadable files included two text files containing lists of geographical coordinate sets that were to be used to test the produced programming. The purpose of this project was multifaceted; it was in part meant to teach me how to implement a greedy search algorithm and its variations. It was also meant to show yet another method of solving the Traveling Salesperson Problem or other problems like it and the method’s strengths and weaknesses – the tradeoffs of computational time and solution optimality.

1. **Approach**

As stated previously, the algorithm in question for this project is a variation of what is traditionally called the greedy search algorithm. A greedy search algorithm is a problem-solving method that opts to go with the best choice at each step, locally. (Wiki) This algorithm **can** produce great results in some circumstances, but not all. Since the choices are being made based on what is best at the local level, repercussions and previous choices alike are not being considered. So, if a tree to represent possible weighted, directed paths to take to get from a StartingCity to an EndingCity looks like this -

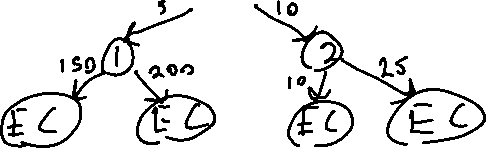


Figure 2.1

- then a greedy algorithm implementation would certainly fail at locating the path of least cost. The greedy algorithm’s depth is limited, so its first choice would be to take the obviously less cost of 5 to City1, even though both final paths from City2 will result in significantly lower total costs.

The greedy approach to the Traveling Salesperson Problem (TSP) could very easily be implemented as the nearest neighbor insertion variation – an approach in which the closest point to the current city is visited next until there are no more cities to visit – but that is not what I was asked to use.

The variation of greedy algorithm being used in this project is the nearest edge insertion heuristic. This is an approach similar in name and process, however there is a slight difference that makes it much more difficult to implement, in my opinion. Instead of choosing cities to visit next based on how close they are to the current city, they are chosen based on how close they are to an edge that is already part of the tour. Then, after all cities have been visited using this heuristic, the tour returns to its starting city to complete the Hamiltonian path.

Figure 2.2 (cs.uu.nl)

A picture containing lamp

Description automatically generated

To implement this algorithm with the given data sets, I decided to use Python to continue the trend from my previous projects in this course. I intend to use this same programming language for each subsequent project, as well. On top of Python’s base functionality, I also opted to enlist the use of a few resources built on top of it. These libraries provided powerful functionality to certain parts of the program and are listed in the references section of this report as well as the beginning of the source code.

I chose the model the relevant data using a City Class that has a location attribute which is a NumPy array whose entries are the x-coordinate then y-coordinate. These cities are read from the file, sifted into a list of City objects, then iterated through and used to determine which will be visited next based on how close they are to an edge. There is also an Edge Class that has starting city, ending city, and length attributes. Cities are popped from their original lists and they and edges appended to the current tour’s lists until there are no more cities to be visited and the Hamiltonian path is completed. Once this tour is completed, the edges’ lengths are summed and the graph is displayed visually using MatPlotLib.

1. **Results** (How well did the algorithm perform?)
   1. **Data** (Describe the data you used.)

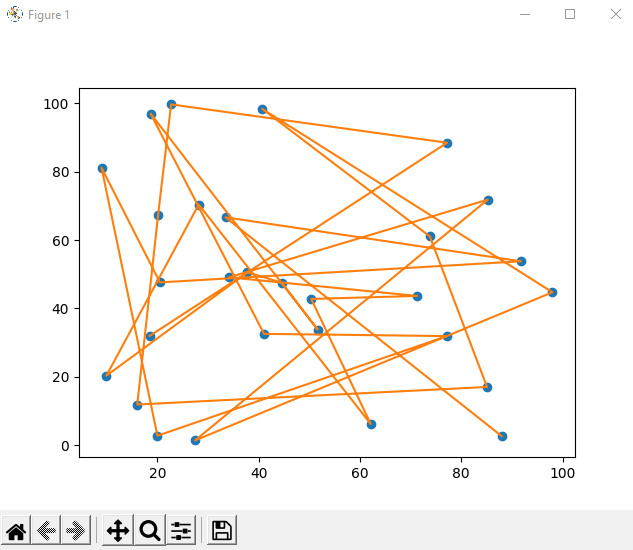
The data used is contained within two plain text files that were provided with the project’s assignment. These text files contain 30 and 40 geographical coordinate sets, respectively, and are read into my program and modeled as mentioned using classes as explained in the previous section. The Euclidean distance formula is used to determine the distance between these cities and therefore the length of the edges. The point-to-line distance formula is used to measure how far each point is from an edge.

* 1. **Results** (Numerical results and any figures or tables.)

Unfortunately, I was only able to successfully check through about ~160 possible edges and add on average 5-6 cities to the tour before the program encountered an error that I was unable to resolve. The error in question was always that a list index one of my loops was trying to access was out of range. I believe the cause to be that my loop setup requires the list of appended edges to grow more quickly than the number of times the loop runs. This was a fundamental flaw in how I designed the algorithm to run, not realizing that edges being added to the tour was inherently a task for outside of the looping structure – or at least it sure seems like it after days of trying to resolve this issue. Despite the lack of a final tour to show, I am happy to say that individual pieces of my program did their jobs well. The graphical user interface (GUI) methods used to display the tour once calculated worked in testing:

Figure 3.2

(unordered list)



The blue dots represent cities, and the orange lines represent the paths between them. (This is an unordered list of cities in the 30-city dataset.) The distance functions were also tested independent of the datasets and return accurate results. City and edge lists and classes were constructed efficiently and handled accurately down to the loop crashing in the algorithm implementation. Most importantly, however, this has greatly helped my understanding of the nearest edge insertion (NEI) technique and its merits. The algorithm’s calculation, search, and comparison of nearly 200 potential edges and placement of 5-6 cities was lightning fast compared to brute force or other slower methods. It took just under a second for those numbers to be crunched before the error was thrown each time. Given 5 cities in roughly .7 seconds, one could assume the time needed to calculate the TSP solution for the 40-city dataset would be roughly 5 seconds. This is incredibly faster than the 2 hours+ that was needed to calculate a solution for merely 12 cities with brute forcing. Although, the solution from NEI may not be as optimal.

1. **Discussion** (Talk about the results you got and answer any specific questions mentioned in the assignment.)

One improvement I could have made to potentially make implementing the algorithm easier is to include dictionaries. A Python dictionary could hold a list of “keys” that are the cities and the “entries” that are the distances to the nearest edge. This was an idea that I started working toward, but ultimately decided against. Maybe it would have been a better choice and I would have more results to show, but it was something I learned nonetheless.

1. **References**

NumPy. (n.d.). Retrieved September 27, 2021, from <https://numpy.org/>

*Pathlib - object-oriented filesystem paths¶*. pathlib - Object-oriented filesystem paths - Python 3.9.7 documentation. (n.d.). Retrieved September 27, 2021, from <https://docs.python.org/3/library/pathlib.html>

*Random - generate pseudo-random numbers¶*. random - Generate pseudo-random numbers - Python 3.9.7 documentation. (n.d.). Retrieved September 27, 2021, from <https://docs.python.org/3/library/random.html>

*Visualization with python¶*. Matplotlib. (n.d.). Retrieved September 27, 2021, from <https://matplotlib.org/>

Wikimedia Foundation. (2021, August 29). *Greedy algorithm*. Wikipedia. Retrieved September 27, 2021, from <https://en.wikipedia.org/wiki/Greedy_algorithm>

[www.cs.uu.nl/docs/vakken/an/an-tsp.ppt](http://www.cs.uu.nl/docs/vakken/an/an-tsp.ppt)