Project 3: TSP – Closest Edge Insertion Heuristic

Hunter Ellenberger

CSE-545-02-4248

Speed School of Engineering

University of Louisville, USA

[hlelle01@louisville.edu](mailto:hlelle01@louisville.edu)

**1.**

During this project I implemented a greedy search algorithm to find the next closest node to the starting node, this was done until all nodes were touched. Finally, I added a return to the first node and inserted that into the tour.

I first did this by constructing a function, named find\_shortest\_path, that finds the closest node to the current one. The function searches through all possible nodes that it could travel to, calculates the distance, then returns both the node that it went to along with the distance it took to do so. This implementation required the import of a variable called maxsize. This variable stores the value of the max integer that is allowed on the current machine. The reason I use this variable is for the first comparison to find the shortest distance between nodes. I need to make sure that the first distance calculated in the function is assigned to the variable shortestDistance, in my case I just make sure there is no possible way that the distance is larger than what its being compared to. The next item I had to import for this function is calc\_distance. This is used to simply compare the distance between the current node and the node the function is calculating to jump to.

As for the implementation of the find\_shortest\_path, it requires three parameters: the node it is starting at, the list of nodes it can jump to, and a dictionary containing the coordinate locations of each node. The function starts by checking if the number of available nodes it can jump to is one, if it is one it returns a list containing two zeros for both the elements, this is so that the distance calculation is not skewed by the final “false jump.” Once this has been checked the function then simply iterates throughout every element in the list of possible nodes to jump to and keeps track of the node which takes the shortest distance to jump to; it also keeps track of this distance. It then returns this pair once it has iterated throughout every element in the list.

The next function I had to implement for this solution was greedy\_insert. This function uses the find\_shortest\_path function to find the shortest path from the current node to the next available node. It then tracks all these tours taken and then returns the final tour that includes all nodes exactly once, excluding the return to the starting node.

To implement these requirements, I started by accepting the following parameters: the starting node and a dictionary containing the nodes and their associated coordinates. I then load all nodes that are in the TSP file into a list that will contain all nodes that have not been visited. Once this is done, there is then a while loop that runs until there are no elements contained in the previously mentioned list. The while loop starts by inserting the “starting” node to the tour. It then calls the find\_shortest\_path function to find the next closes node along with the associated distance. The while loop then removes the original start node from the list of possible nodes to visit. Next the start node variable is then reassigned to the previously found closest node. Finally, the while loop adds the distance from the previous node to the now start node to a variable that contains the total distance traveled in the tour. There is then a small segment of code used to calculate and add the distance from the last node visited to the first node in the tour, completing the Hamiltonian Cycle.

The final function I had to implement, named plotter, was used to produce the GUI that shows both the nodes that are specified in the TSP problem and the tour that is traversed by the algorithm.

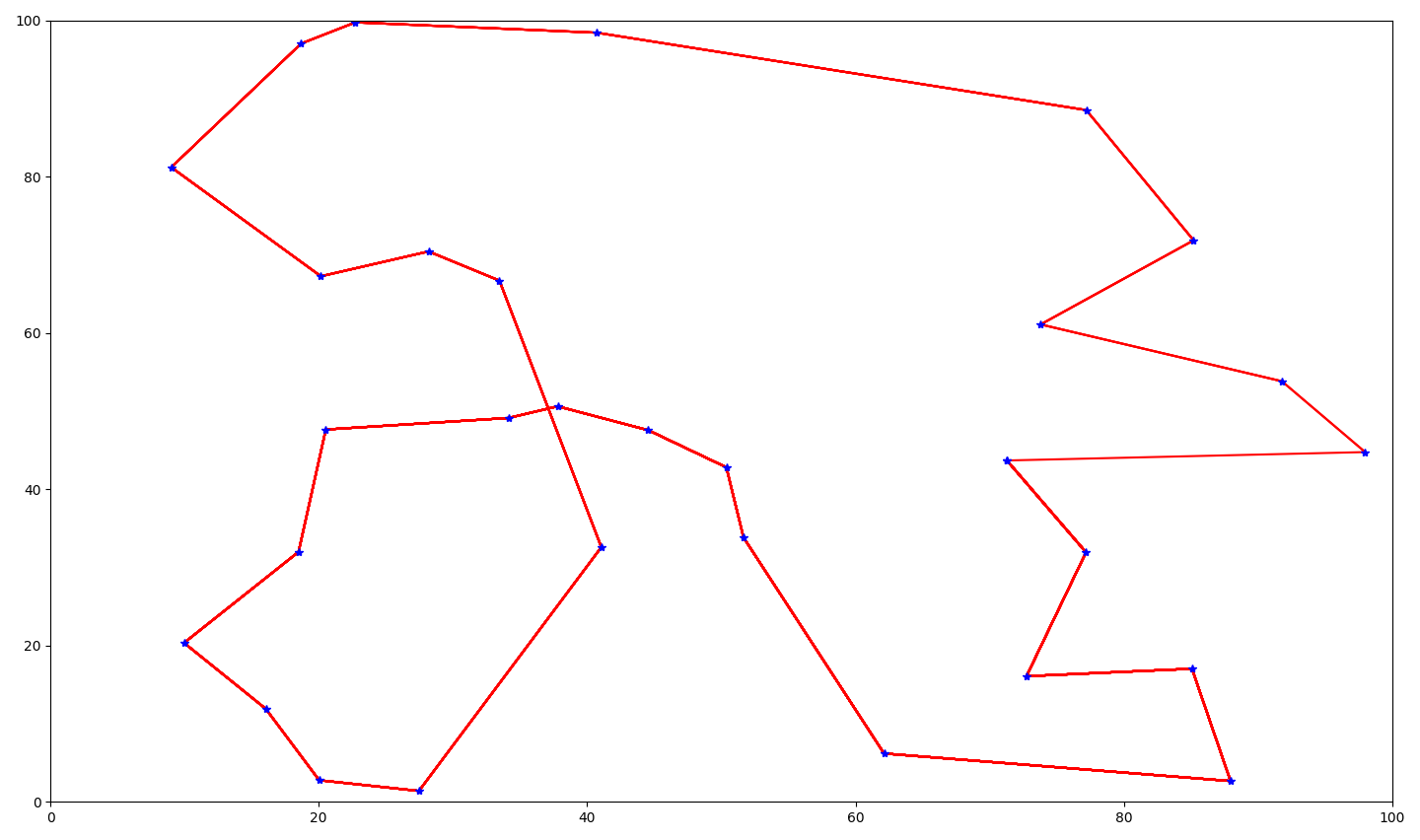
I started implementing this by specifying the following parameters: a dictionary that contains the nodes along with their coordinates and the tour that is taken. In this function I start by appending the x and y coordinates to two separate lists, each containing the respective type of coordinate. I then plot the lines using a for loop that iterates through each element, it connects the current element it's on with the next element in the tour. I then plot the points where the nodes are located and finally, I plot the graph.

**2.**

The type of algorithm used for this project is a greedy algorithm. There are many different implementations of this type of algorithm. However, the throughline is that this type of algorithm always goes for the locally optimal solution, even if this results in a non-optimal final solution; on the upside this usually results in less compute time due to not having to consider the long term. In my specific implementation my algorithm will look one step ahead, considering all the possible nodes it can jump to and then picking the one with the shortest distance from the current node. This algorithm then inserts this picked node into the tour and repeats the process until no nodes are left.

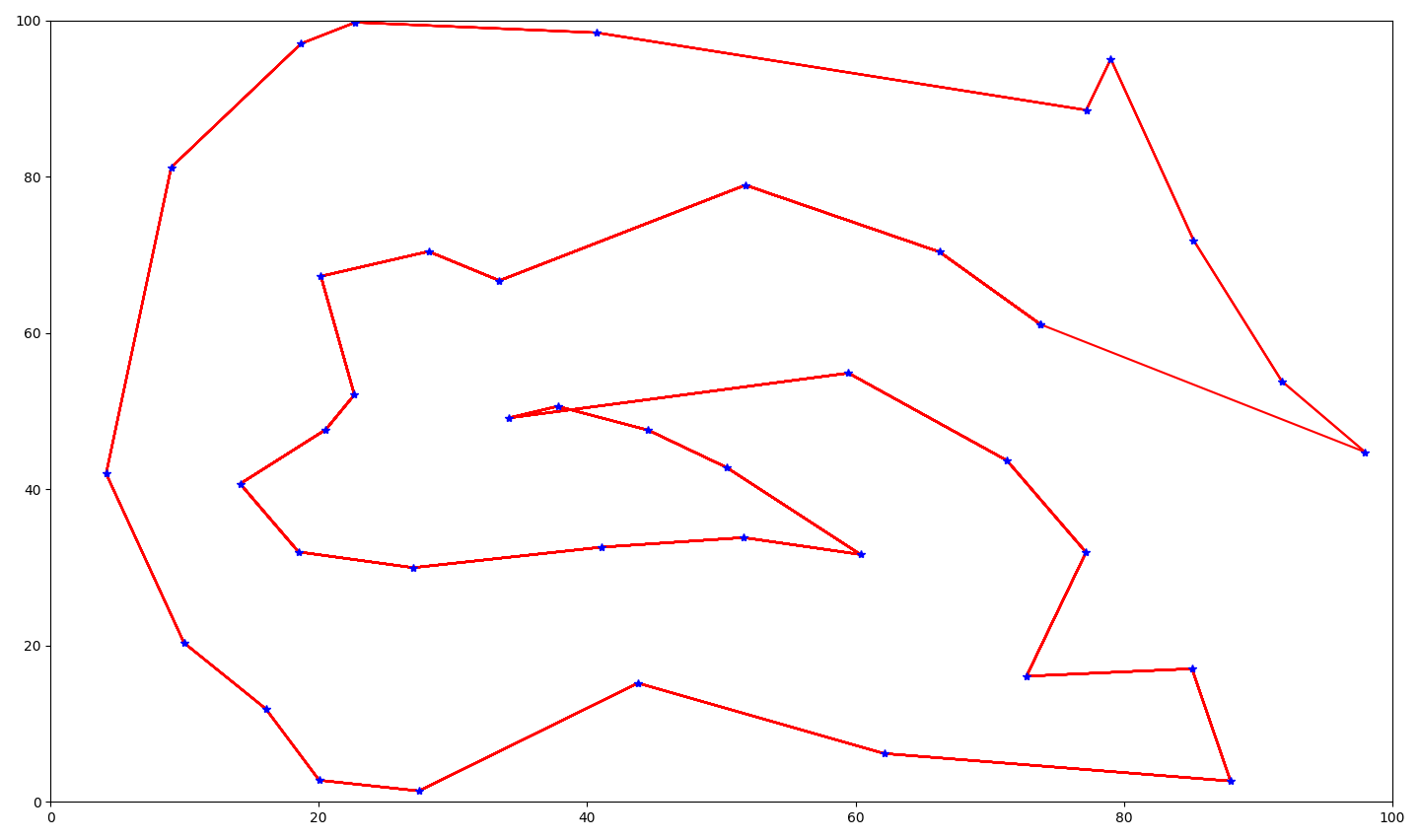
**3.**

In this project there were two files provided, both being TSP files. One of the files contained thirty points, the other contained forty points. As for runtimes the file containing thirty points took around 261ms whereas the file with forty points took around 286ms. Compared to the BFS&DFS algorithms this algorithm takes much longer, somewhere between ten and twenty times. However, the search area is much longer and with my implementation it runs the algorithm a total of n times, where n is the number of nodes in the file (will be expanded further in Part 4). With this in mind, the algorithm definetly holds up to its predecessors. Especially when it is compared to the brute force algorithm, an algorithm that could not even reasonably compute more than eleven nodes on my system.

As for tours the file with thirty nodes ended up being a distance of 484.475 and took the path 17 -> 7 -> 30 -> 24 -> 1 -> 15 -> 10 -> 16 -> 11 -> 12 -> 18 -> 4 -> 29 -> 13 -> 25 -> 6 -> 20 -> 8 -> 2 -> 14 -> 26 -> 5 -> 9 -> 27 -> 22 -> 28 -> 19 -> 23 -> 3 -> 21 -> 17. The graph for this path can be seen below:

Seeing as there is only one intersection seen in this graph the path seems pretty optimal, although it is not the most.

The tour for the file with forty nodes ended up having a distance of 607.854 and took the path 23 -> 34 -> 36 -> 2 -> 14 -> 26 -> 35 -> 4 -> 33 -> 29 -> 38 -> 8 -> 10 -> 40 -> 16 -> 11 -> 12 -> 18 -> 37 -> 17 -> 7 -> 30 -> 24 -> 1 -> 15 -> 39 -> 20 -> 6 -> 25 -> 13 -> 31 -> 5 -> 9 -> 27 -> 22 -> 28 -> 32 -> 19 -> 3 -> 21 -> 23. The graph for this tour can be seen below:



**4.**

I was proud of the results I got seeing as they seem to be relatively close to the result that would be optimal for these specific TSP problems. As I previously mentioned, my algorithm simply looks ahead one layer to see which node to jump too next. However, I did end up adding a final step that would choose the most optimal starting node. I did this by running the algorithm for each node to see which one resulted in the shortest distance, this does increase the runtime by quite a lot but nowhere enough to be considered non-computable in this case. There are many ways I could improve this algorithm such as expanding that local range to consider two steps ahead instead of just one, but for the compute used I believe this implementation is perfectly fine.

**5.**

The only resources I needed to use were related to constructing the GUI. In this version I went with matplotlib to plot the graph, and the following resources were used:

[**https://stackoverflow.com/questions/35363444/plotting-lines-connecting-points**](https://stackoverflow.com/questions/35363444/plotting-lines-connecting-points)

[**https://www.tutorialspoint.com/showing-points-coordinates-in-a-plot-in-python-using-matplotlib**](https://www.tutorialspoint.com/showing-points-coordinates-in-a-plot-in-python-using-matplotlib)