Project 3: Hamiltonian Circuit with Greedy Insertion

Richard Douglas
M.S. Computer Science
Speed School of Engineering
University of Louisville, USA
rcdoug@louisville.edu

1. Introduction

1.1 Goals:

The learning objective in this project is to implement a greedy insertion algorithm to find the best path for a hamiltonian circuit in a graph.
The algorithm should establish a starting circuit containing three nodes, and insert a node in between the two vertices that create the edge closest to that node.
We were also tasked with creating a GUI to visualize the algorithm. This proved VERY useful in understanding whether a solution was good or not, and actually was instructive on how to better solve the problem.

2. Approach

2.1 Concept

This algorithm was surprisingly tricky to implement. There are various factors to consider - which first three nodes to select, how to select a next vertex to insert into the current list of visited vertices, where to insert that node.
As it turns out, all of these factors do matter in creating this algorithm. I experimented with various approaches before landing on one that worked well. We know that if two edges cross, the algorithm is not working as well as it could. This was shown in the GUI as I worked on the algorithm.
The trickiest part was how to insert nodes in the correct position.

2.2 Implementation

2.2.1.1 Greedy insertion

2.2.1.1.1 Choosing a start node:

I simply chose a start node that was as deep as I could in the corner of the graph. In other words, the one that had the farthest from the x1,y1 coordinates (0,0), measured with the line formula L = sqrt((x2-x)^2-(y2-y1)^2), with x2,y2 = (0,0). That way, I would avoid errant paths that would be difficult for the insertion algorithm to reconcile.

2.2.1.1.2 Choosing the other two start nodes:
This was simple. I just chose the two closest nodes to the start node. With those two nodes being n1, n2 respectively, the starting path would be [start, n1, n2, start].

2.2.1.1.3 Selecting a node to insert:
This wasn't too hard to decide, although I am considering options to make this algorithm run in a better time efficiency than O(n^2).
First, I set up the initial path as described above. I also initialized a list of all unvisited nodes… this is an alternative to the previous hash map of all nodes for visited, where a boolean value represents whether is has been visited.
Finally, I enter a while loop, with the condition that the length of the "visited" list is less than the amount of nodes in the graph + 1.
In the while loop, a call is made to the greedy insertion algorithm. Each visited node pair is explored in a for loop, but the nodes before and after these two (by index, or order in the path) are also tracked.
Nested inside this for loop is another, which sums the cost of the current node pair to each of the unvisited nodes. if this sum is less than the current least sum, this node is set as the node to append to our list, and its insertion options are explored. Thus, we choose the closest node to the graph. I chose this approach because of the way I chose to pick an insertion location, although I admit that it seems to be a little contrived. We very well may only need to select the closest node to any point in the graph to achieve the same result

2.2.1.1.4 Choosing where to insert that node:
Recall that I tracked the nodes before and after the node pair in the nested for loop in this algorithm.
I did this so that I could tell if a line drawn to either of these points from the selected insertion node intersects with the

line between that node and the first and second node in our node pair respectively. If so, this would be the insertion location, otherwise, the insertion location would be between the node pair examined.
To clarify, say that we have the four nodes n0,n1,n2,n3 with respective positions (1,1),(1,6),(6,6),(6,1) and we selected to insert the node n_ins at position (0,5) because it is closest to the sum of its distance between n1,n2. We would not want to insert it between those two nodes, because the line segment (n_ins,n3) would intersect with the line segment (n0,n1). Thus, we have determined its appropriate insertion location is actually between n0 and n1. I.e. we have found the line segment nearest the insertion node and inserted it there.

2.2.1.1.5 GUI
I used tkinter and matplotlib to display the graph, and created a method for visualizing the path by displaying edges in a current path. This animation function is called on a GUI object for every iteration of the algorithm.

3. **Results** (How well did the algorithm perform?)
This algorithm performs a time complexity a little worse than that of the greedy algorithm implemented in project 1, where a path is traversed back to a start node by simply selecting the nearest unvisited node and then appending that node to the path.
The greedy insertion method, however, usually outperforms the previous greedy algorithm on finding an efficient traversal. It does this because the greedy appending algorithm is quite liable to select closest nodes until only crosses remain from a given node, whereas we have avoided crosses in greedy insertion. That really depends on the graph, however.

3.1 Data (Describe the data you used.) :

3.1.1 TSP Files: I used the .tsp files provided in the .zip file Professor Yampolskiy provided.

3.1.2 Logging results: the visual representation of my results were created with the logs I saved in .csv format from the results of running the algorithms I wrote on the .tsp files. The following are written to the logs in each row. you can check out the results in the csv file:

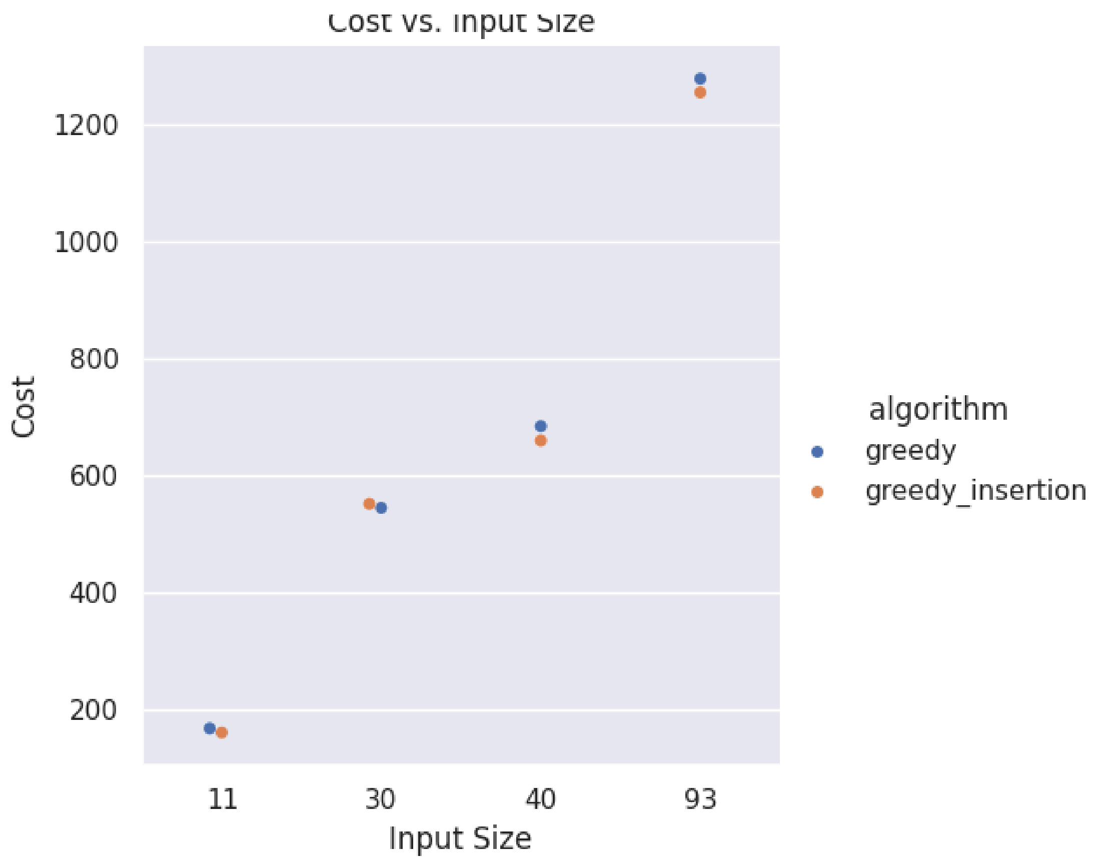
3.1.3 algorithm,input_size,best_cost,best_cost_path,least_hops,least_hops_path,runtime,tsp_file

3.1.3.1 Algorithm: which algorithm I used
3.1.3.2 Input size: on the iteration for
3.1.3.3 best_cost: the best cost calculated by the algorithm implemented
3.1.3.4 best_cost_path: path with the best cost to reach the target
3.1.3.5 least_hops: least amount of hops to reach the target
3.1.3.6 least_hops_path: path corresponding to least hops
3.1.3.7 Runtime: The measured runtime calculated by the difference between a timestamp taken immediately before the algorithm starts, and one taken immediately after it completes (end - start).
3.1.3.8 File: the TSP file used in making the calculation. Although it isn't germain to this report, it was useful for debugging.

3.2 Results (Numerical results and any figures or tables.) :

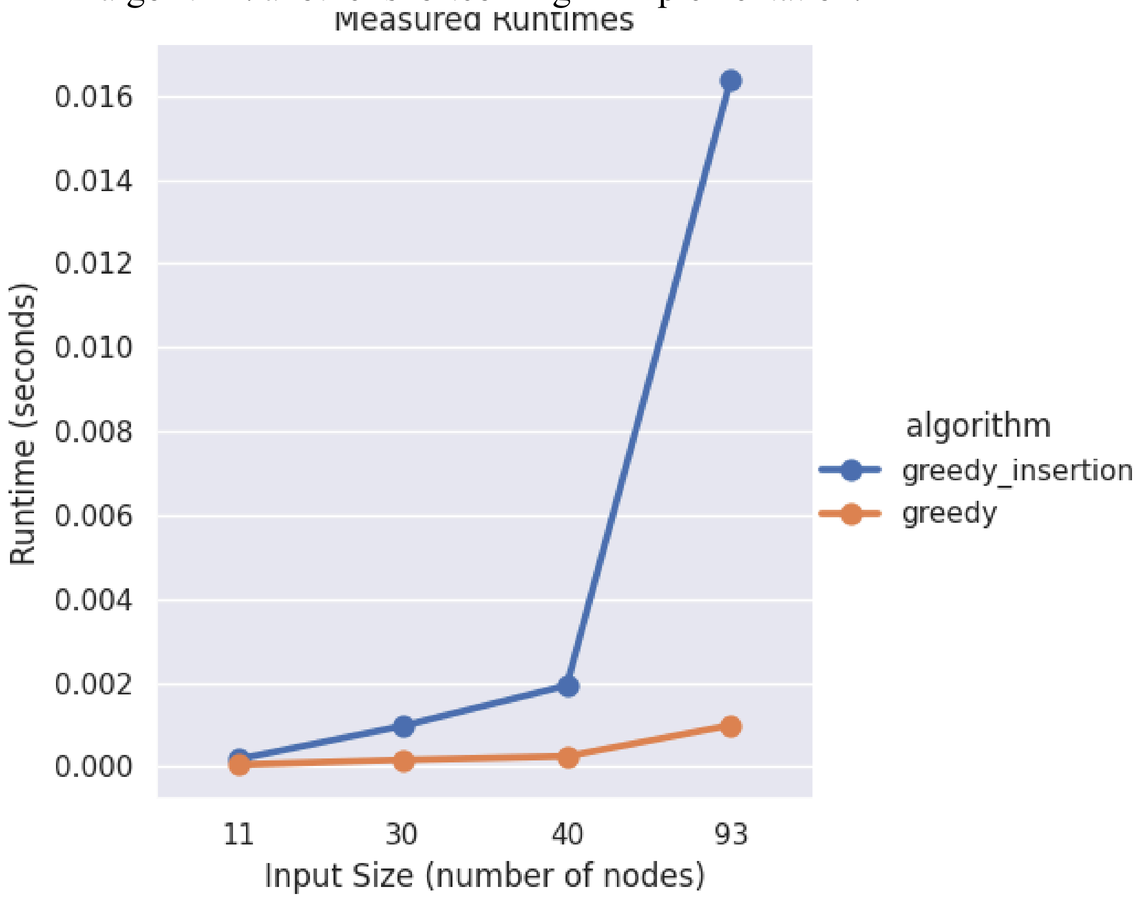
3.2.1 Figure 1: Calculated best path results

This is a categorical box plot of the best costs calculated. I used the greedy appending (greedy), and greedy insertion algorithms to traverse the graph. Unfortunately, with this algorithm, the greedy appending algorithm found a better path in the Random30.tsp graph. I believe this is because the nodes are near one another, and there is a bottleneck that happens in the middle of the graph where edges are appended to that are near one another, but the best path is actually one that traverses a wider path. It may be avoidable by selecting the weighted center of the graph, then the node nearest that point as the start node.



3.2.2 Figure 2: Runtime results

Here we compare the stochastic runtimes of these algorithms. Clearly, the greedy insertion algorithm is much slower than the greedy algorithm. another shortcoming in implementation.



Video of visualization:

You can watch the implemented algorithms run by running the driver file, TSP_driver.py

4. Discussion

I would still like to make efforts to improve my GUI. Currently, it does visualize the algorithms but I want to save the frames so that they can be scrolled through to examine them. I am also curious as to how I might decrease the time complexity of the greedy insertion method. I wonder if I can improve by doing fewer calculations on each iteration. the nested looping certainly adds time complexity, although I am not entirely certain how to implement the described algorithm otherwise.

Also, the python files I have built up are becoming unwieldy. I need to shake off the code that I don't use in future problems, if we continue to work on tsp problems.

5. References

on directed graphs:
https://www.whitman.edu/mathematics/cgt_online/book/section05.11.html#:~:text=A%20directed%20graph%2C%20also%20called,or%20(w%2Cv)

algorithms:
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
https://www.geeksforgeeks.org/a-search-algorithm/

python documentation:
https://seaborn.pydata.org/generated/seaborn.catplot.html
https://docs.python.org/3/library/sys.html
https://docs.python.org/3/library/os.html
https://docs.python.org/3/library/datetime.html
https://docs.python.org/3/library/gc.html
https://docs.python.org/3/library/csv.html
https://matplotlib.org/stable/index.html
https://pandas.pydata.org/docs/index.html
https://docs.python.org/3/library/math.html
https://docs.python.org/3/library/itertools.html#itertools.permutations
https://docs.python.org/3/library/random.html