Project 3: Traveling Salesman Problem – Closest Edge Insertion Heuristic

Nick York
CES Department
Speed School of Engineering
University of Louisville, USA
neyork01@louisville.edu

1. Introduction

The Traveling Salesman Problem ("the TSP") was introduced in Project 1. A quick recap: the TSP requires finding the minimum cost route ("best route") for a salesman to visit a list of cities without visiting a city more than once, except for the starting city. The salesman must return to their starting city at the end. Each city is represented by an xy-coordinate pair and numbered starting at one. The coordinates are provided by a .tsp file ("TSP file"). The cost to travel between two cities is the Euclidean distance between them, calculated as:

$$cost = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

Project 3 introduces the closest-edge insertion heuristic ("the heuristic"), a variant of the greedy heuristic. The greedy heuristic makes the locally optimal choice without considering future choices. The closest-edge insertion heuristic works by building a Eulerian cycle ("tour") that visits each edge exactly once. The goal of this project was to write a program ("the program") to find a tour with the minimum cost ("the optimal tour") for a given set of nodes using the heuristic.

In terms of Project 3, cities are nodes, connections are edges, and the route is a tour. Any node can connect to any other node. The heuristic runs once for each possible starting node. Initially, there are no edges. An edge is created between the starting node and the closest node to it. Then, the heuristic inserts nodes closest to an existing edge in the tour. The edge with the closest node is removed (if there are three or more nodes in the tour) and two new edges are created from its endpoints. The tour with the minimum cost is the optimal tour.

2. Approach

I wrote a Python program that allows the user to specify which TSP file to use. Next, the program reads and parses the xy-coordinate pairs in the TSP file. To implement the heuristic, I repurposed the Map and City classes from my Project 2 program to represent the TSP. I renamed Map to Network and City to Node to abstract them for future use cases.

The constructor for Network takes a list of coordinates and uses them to build a dictionary of Node objects assigned to *nodes*.[1] A copy of *nodes* is assigned to *available_nodes* to track which can be inserted. Nodes inserted into the tour, edges formed between those nodes, and the cost of the tour are stored in *nodes_in_tour* (initially empty), *edges_in_tour* (initially empty), and *cost* (initially set to 0), respectively. Network also stores the optimal tour and its cost across iterations of the heuristic using *optimal_nodes* (initially empty), *optimal_edges* (initially empty) and *optimal_cost* (initially set to positive infinity).

I also created two new classes for this project: Edge and Triangle. Edge represents edges formed by two nodes. An Edge is created by passing two nodes, node1 and node2, and assigning them to *node1* and *node2*. Then, a helper function is called to find the edge's slope m using

$$m = \frac{y_2 - y_1}{x_2 - x_1} \quad (2)$$

where (x_1, y_1) is *node1*'s coordinates and (x_2, y_2) is *node2*'s coordinates. The value of m is stored as *m*. Next, a second helper function is called to find the Edge's length using

$$length = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (3)$$

where (x_1, y_1) is *node1*'s coordinates and (x_2, y_2) is *node2*'s coordinates. Finally, *closest_node* (initially empty) and *closest_node_dist* (initially set to positive infinity) track the closest node and the distance to it.

Triangle represents a triangle constructed from an edge and a node. Triangle is used to solve for the distance between the edge and node used to create it. For example, let the edge be AB, whose nodes are A and B, and the node C. If we create edges CA and BC, then we can solve for angles CAB and BCA. If either angle is greater than 90°, then the shortest of CA and BC is the minimum distance between AB and C (Fig 1). If neither angle is greater than 90°, then the minimum distance is the length of the line perpendicular to AB and C

$$length = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \quad (4)$$

where (x_1, y_1) is A, (x_2, y_2) is B, and (x_0, y_0) is C (Fig 2).

[1] Class attributes are italicized to differentiate them from regular variables.

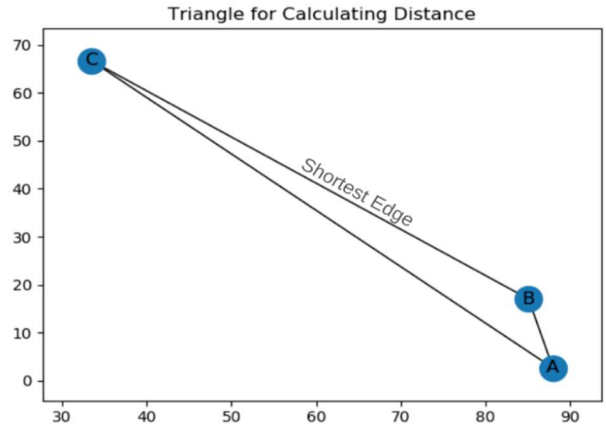

Fig 1. A triangle whose ABC angle is obtuse. BC is shorter than CA and its length will be the shortest distance.

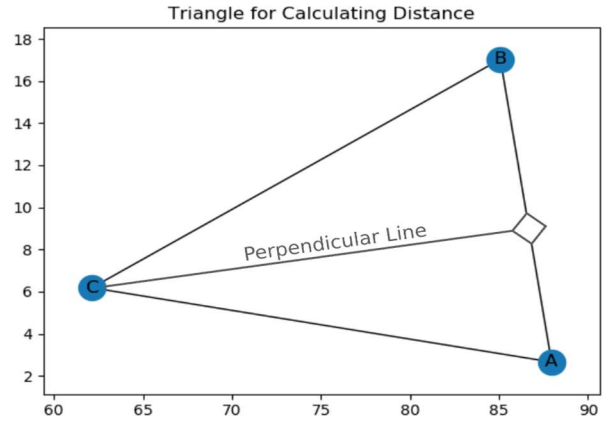

Fig 2. A triangle whose ABC and CAB angles are acute. The length of the perpendicular line passing through AB and C will be the shortest distance.

The heuristic is implemented using two functions: iteration() and insertion(). iteration() iterates *n* times over insertion(), where *n* is the number of nodes in the Network. Each iteration picks a different node to start from. After insertion(), if *cost* is less than *optimal_cost*, then *optimal_cost* is updated to *cost*, *optimal_nodes* is updated to a copy of *nodes_in_tour*, and *optimal_edges* is updated to a copy of *edges_in_tour*. Afterward, *available_nodes* is updated to a copy of *nodes* (to reset the nodes to insert), *nodes_in_tour* updated to an empty list, *edges_in_tour* updated to an empty list, and *cost* to 0. Updating the prior values resets the Network object perform another iteration of the heuristic. After each node has been used as a starting node, the final attribute values are the optimal solution.

Meanwhile, insertion() implements the core logic of the heuristic based on the starting_node its passed. insertion() calls find_first_edge() and passes starting_node. find_first_edge() inserts the starting_node into *nodes_in_tour* and iterates over *available_nodes* to find the closest node to starting_node. The closest node is inserted and an edge is created from both nodes. When the function returns, insertion() continues by iterating over each edge in

edges_in_tour (outer for loop) and each node in *available_nodes* (inner for loop) while *available_nodes* is not empty.

Each iteration of the outer loop finds the closest node to any edge. Each iteration of the inner loop creates a Triangle from the current edge and current node and finds the distance between them to use as min_distance. Then, min_distance is compared to the current edge's *closest_node_dist*. If the min_distance is less than *closest_node_dist*, then *closest_node_dist* is updated to the min_distance and *closest_node* is updated to the current node. This is repeated till no nodes remain, at which point the function exits the inner for loop. The final values *closest_node* and *closest_node_dist* are used. If this is the first iteration of the outer loop, then the current edge is assigned to optimal_edge. Otherwise, optimal_edge exists and the current edge's *closest_node_dist* is compared to optimal_edge's *closest_node_dist*. If the former is smaller, then the current edge is assigned to optimal_edge. Else if their distances are equal, then tiebreaker() is called and the two edges are passed as arguments.

tiebreaker() takes two arguments, edge1 and edge2. The function calculates the current cost based on *edges_in_tour* using equation 1 and summing the results. The length of edge1 is subtracted from the current cost. Then, the endpoints of edge1 and its *closest_node* are used to create two new edges and their length is added to the current cost. The new current cost reflects inserting edge1's *closest_node* into the tour. The process is repeated for edge2 and both costs compared. If edge1's cost is less than or equal to edge2's, then edge1 is returned. Else, edge2 is returned. The returned edge is assigned to optimal_edge. This is repeated till no edges remain, at which point the function exits the outer for loop. The final value of optimal_edge is used and its *closest_node* inserted into the tour and edges created using it and the endpoints of optimal_edge. If needed, the optimal_edge is removed to maintain the tour.

The attributes *closest_node* and *closest_node_dist* for each edge in *edges_in_tour* are reset to their initial values and optimal_edge is set to None to prepare for the next iteration of the while loop. To remove nodes from *available_nodes*, the program removes the node to be inserted after each iteration of the outer loop. The number of edges in *edges_in_tour* grows as the number of nodes in *available_nodes* shrinks. Once no more nodes are available, the function exits the while loop, calculates the final cost for the tour based on *edges_in_tour*, and returns.

3.1 Data

For this project the students were provided two TSP files, Random30.tsp with thirty coordinates and Random40.tsp with forty coordinates, for us to test our program on.

3.2 Results

The results of the heuristic are written to a CSV file under a folder named Results. Fig 3 lists the results of the heuristic for Random30.tsp and Random40.tsp taken from their respective CSV files. Fig 4 demonstrates how the heuristic builds the tour progressively as nodes are inserted. The optimal route for each TSP file is plotted and saved as a PNG file under Results (Fig 5) and outputs the solution to a Matplotlib GUI (Fig 6).

TSP File	Random30	Random40
Optimal Tour	[9, 27, 22, 5, 26, 14, 2, 12, 18, 11, 16, 10, 8, 4, 29, 13, 25, 6, 20, 17, 23, 7, 19, 28, 30, 24, 15, 1, 3, 21]	[4, 35, 33, 29, 38, 31, 18, 12, 11, 16, 10, 8, 40, 13, 25, 6, 20, 37, 26, 14, 2, 23, 34, 19, 17, 7, 30, 24, 15, 1, 36, 5, 28, 32, 39, 9, 27, 22, 3, 21]
Edges (in order of insertion)	[(9, 27), (27, 22), (9, 5), (5, 26), (26, 14), (14, 2), (12, 18), (12, 11), (16, 10), (11, 8), (10, 8), (2, 4), (18, 29), (4, 13), (13, 25), (25, 6), (29, 20), (6, 20), (17, 23), (17, 7), (22, 28), (19, 28), (7, 24), (16, 15), (30, 15), (30, 1), (24, 1), (19, 3), (23, 21), (3, 21)]	[(4, 35), (4, 33), (29, 38), (33, 31), (18, 12), (12, 11), (11, 16), (10, 8), (10, 40), (31, 13), (13, 25), (25, 6), (29, 20), (6, 20), (16, 37), (35, 26), (18, 2), (14, 2), (23, 19), (17, 7), (7, 24), (40, 15), (30, 15), (30, 1), (24, 1), (37, 36), (34, 36), (26, 5), (34, 28), (19, 32), (28, 32), (38, 39), (8, 39), (5, 9), (9, 27), (14, 22), (27, 22), (23, 3), (17, 21), (3, 21)]
Cost	507.3310677	610.1403428
Running Time (sec)	2.808269	8.875364

Fig 3. Results for each TSP file provided by the professor.

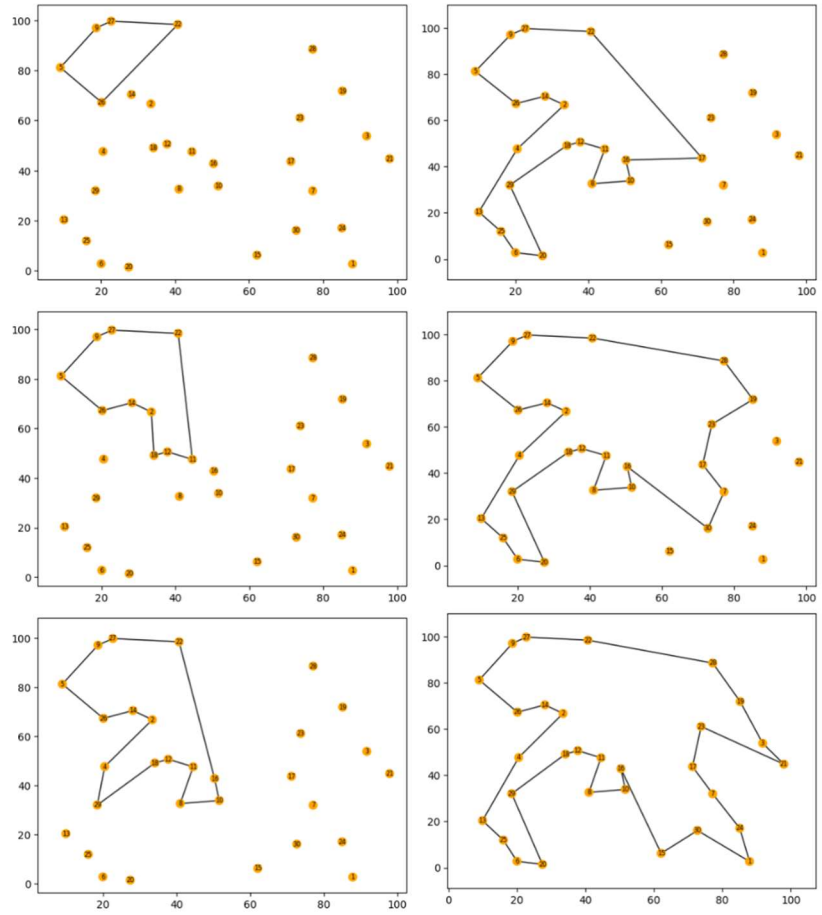

Fig 4. Progression of the heuristic for the Random30 optimal tour.

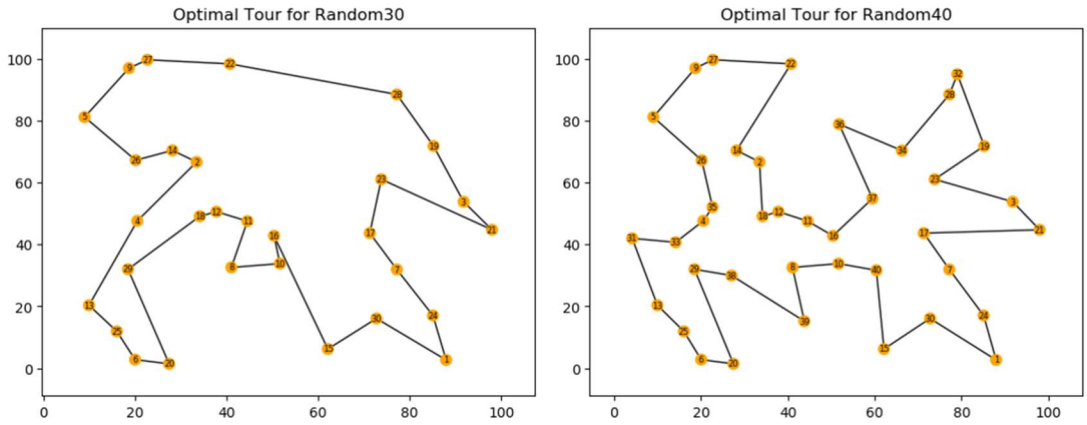

Fig 5. Graphs showing the optimal tour for Random30 and Random40.

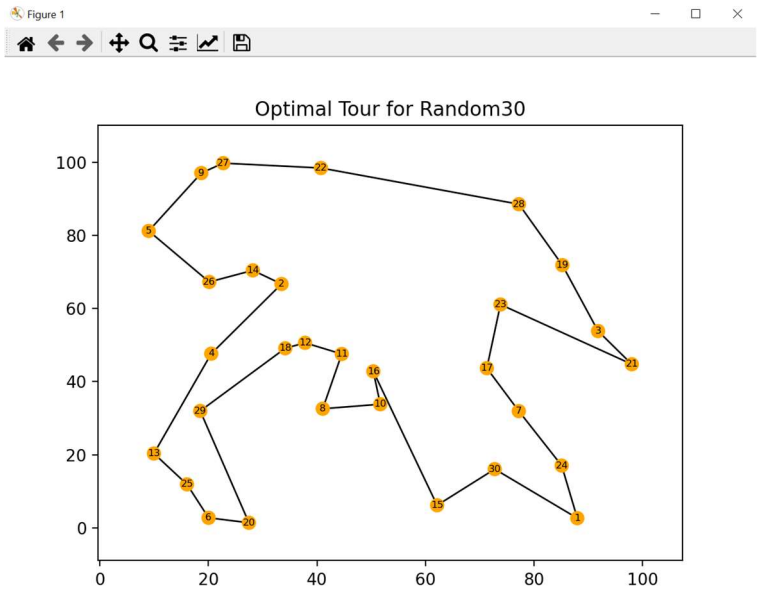

Fig 6. Matplotlib GUI for viewing solution to Random30.

Discussion

One of the challenges of the heuristic was selecting the starting node. My solution was to run the heuristic with each node as a starting node once. After each iteration I compared the most recent tour with the current optimal tour and kept the better one. However, this would be very inefficient for a larger number of nodes. The other challenge was selecting which node to insert next. Specifically, the difficult part was finding the distance between a node and an edge.

The reference provided by the professor to the equation from a point to a line was useful, but only for nodes where a perpendicular line could be drawn between it and the edge under consideration. The reason is that the edge is bounded by its endpoints, but the equation describes an infinite line intersecting those points. The solution was to apply geometry and trigonometry. A perpendicular line makes an angle of 90° with the line it intersects. If the angle between the point and the edge were 90° or less, then a perpendicular line would intersect the edge itself. Otherwise, the distance to the node could be found from the length of the shortest line from each endpoint of the edge to the node. The inclusion of three nodes favored a triangle, whose edges could be used to find both angles and lengths.

The closest-edge insertion heuristic is far faster and more memory-efficient than the brute-force approach; the heuristic finds the optimal tour for 40 cities in roughly 9 seconds, whereas the brute-force approach would both never finish and run out of memory long before. Although BFS and DFS are far quicker (fractions of a second), the search algorithms only find an optimal route from a start point to an end point, and the route does not include all cities. The heuristic finds a tour which solves the traditional TSP (visiting each city exactly once except the starting city as the ending city).

The heuristic's simplicity is one of its advantages, as it is easy to comprehend and code (as opposed to a recursive algorithm). However, that simplicity hurts it - the heuristic always chooses the locally optimal choice without regard to future choices. One or more instances in the TSP files

may exist where a locally non-optimal choice leads to a better tour than the solutions in Fig. 5. Another constraint may be memory. Network must store every node to consider and make copies of those nodes to store in *available_nodes* and *nodes_in_tours*. Likewise, Network stores the Edges in *edges_in_tour* and the current optimal tour across iterations of the insertion(). On a related note, copying those data structures may hurt the heuristic for a larger number of nodes. Even if they were not copied, the alternative is to iterate over *nodes* and compare those with nodes in *nodes_in_tour*, which is also not ideal.

References

Wolfram MathWorld Entry on Eulerian Cycle

Clark University, Summary of trigonometric identities