Nearest Insertion TSP

Emily Elzinga and Chris Blaser

December 2022

1 Abstract

In this report, we detail two algorithms that provide heuristic solutions to the Traveling Salesman Problem: Greedy and Nearest Insertion. The Greedy solution is only used as a benchmark heuristic to determine the quality of the Nearest Insertion algorithm. The complexity of the Greedy algorithm on a typical TSP problem is $O(n^2 log(n))$. The complexity of our algorithm is $O(n^3)$, but it will never error on asymmetric graphs with infinite distances between given points. The implementation of the Greedy algorithm will return a null solution for asymmetric and/or graphs with infinite distance between edges.

2 Introduction and Description of the Problem

The Traveling Salesman Problem has a deceptively simple description: Given a graph with either symmetric or asymmetric weighted edges connecting a set of nodes, what is the shortest path from a starting node that visits all the nodes exactly once and returns to the start node? For a large problem, verifying whether a given path an optimal is extremely hard for a computer to do in a single lifetime. This is because the set of possible solutions increases exponentially as the number of cities increases.

3 Algorithm Description

The nearest insertion heuristic is an approximation algorithm for the traveling salesman problem that attempts to insert the closest unvisited city to the partial tour so far [1]. Thus, it must check every city it has already visited for the closest unvisited city and compare each of the closest city options to find the overall closest city to the partial tour. Once found, the algorithm then attempts to find the spot in the partial tour where inserting this unvisited city would increase the overall cost of the tour by the least amount. It then inserts the city between the two visited cities in the partial tour. This is repeated until the partial tour is a complete tour. Thus, the algorithm consists of 3 main parts:

- 1 Initialize a partial tour. Determine the first 3 cities to create the partial tour. Choose the initial city, and greedily determine the second city by choosing the closest city to the start city. From the second, again greedily choose the closest city that creates a partial tour.

- 2 Selecting the next (nearest) city. Find cities k and j where k is an unvisited city and j is a city in the partial tour so far and for which the edge from j to k is minimized.
- 3 Inserting the city. Find the edge e_{ij} in the partial tour that minimizes e_{ik} + e_{kj} - e_{ij}. Once found, insert city k between city i and city j. Because this implementation needs to solve asymmetric graphs, edge direction matters.
- 4. Continue repeating steps 2 and 3 until a full tour is found.

The benefit of using the nearest insertion method is that it is a very fast method of getting a solution. Theoretically, it will only return answers no worse than double to optimal tour. If a "decent" solution is desired, this is a fast way to get it. The downside is that although branch and bound will eventually get the optimal solution, in larger sets of cities, it is unlikely the solution from nearest insertion will be optimal. It is therefore a "quicker and dirtier" greedy algorithm equivalent. We chose this algorithm because we thought it would be interesting to contrast the runtime and solution quality of Branch and Bound versus a heuristic like Nearest Insertion.

The structure of our algorithm came from Heuristics and Local Search by Jose Oliveira and Maria Carravilla [1] in which they outline the basic structure of several heuristic algorithms for approximating the traveling salesman problem. This heuristic as is is very quick and is a pretty good approximation for symmetric graphs with an approximation factor of 2. In all of the literature we read on the subject, we never saw a change to the algorithm for graphs, but for asymmetric graphs, we knew we would need to make some changes. Since we were testing with asymmetric graphs, we needed a way to ensure that, if it returned a full tour, the tour was actually valid. In order for this method to solve an asymmetric graph, we had to apply some checks on missing edges and possible partial tours to the algorithm to ensure that an actual tour is found every time.

4 Complexity

Because there is an outer while loop that only stops when all the cities have been visited, a for loop inside that one to iterate through all the cities in the route to, and one inside the inner for loop to iterate through all the unvisited cities, our implementation of the Nearest Insertion Algorithm is $O(n^3)$. According to the literature, the Nearest Insertion algorithm is $O(n^2)$ because it doesn't iterate through the visited cities, but we were unsure how to make it that simple without sacrificing accuracy.

5 Analysis of Results

Of all the algorithms that we tested (Random Permutation, Greedy, Branch and Bound, and Nearest Insertion), Nearest Insertion proved to be the most reliable in determining a reasonably good solution in a short amount of time. For 500 cities, Nearest Insertion converged in just 40 seconds. Although there were no other heuristics to compare against, the fact that it even converged when the others quit before 200 cities speaks for itself. Tables 1, 2, and 3 show a comparison of the runtimes and solutions for the different algorithms.

Table 1 – Runtime statistics for the random algorithm

	Random	
# Cities	*Time (sec)*	*Path Length*
15	0.0035962	19761
30	0.0211878	38361.8
60	21.422	82784
100	TB	TB
200	TB	TB
17	0.00262	24601.4
500	TB	TB

Table 2 – Runtime statistics for the Greedy algorithm

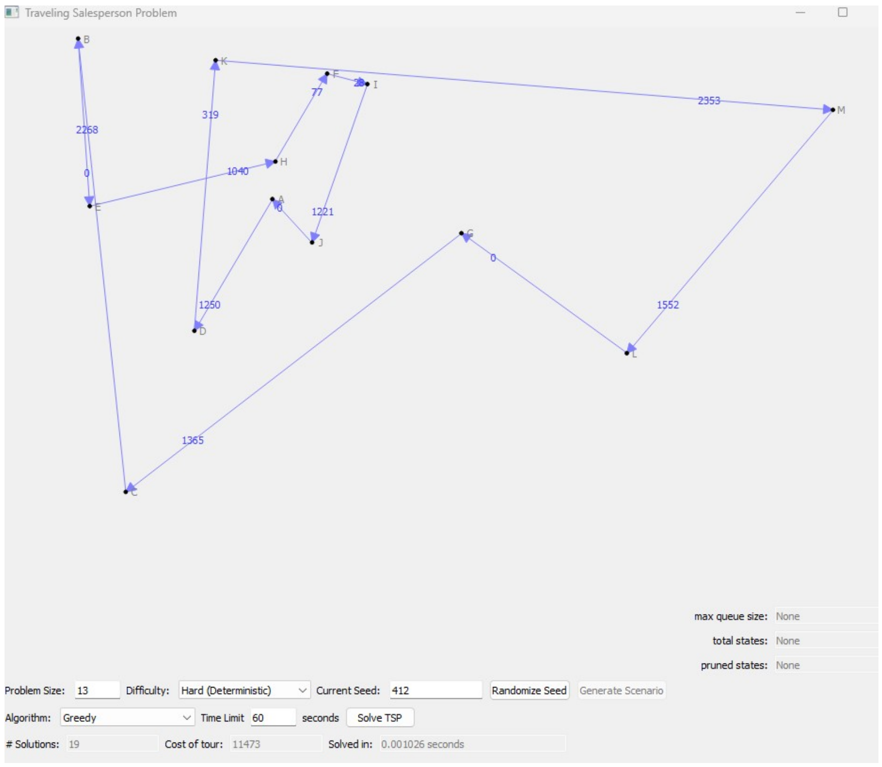
	Greedy	
# Cities	*Path Length*	*% of Random*
15	15181.6	76.83
30	19821.4	51.67
60	36385.4	43.95
100	35187	N/A
200	TB	TB
17	18544	75.38
500	TB	N/A

Table 3 – Runtime statistics for the BB algorithm

	Branch and Bound		
# Cities	*Time (sec)*	*Path Length*	*% of Greedy*
15	20.62142	9090	59.88
30	TB	TB	TB
60	TB	TB	TB
100	TB	TB	TB
200	TB	TB	TB
17	108.3128	10899	58.77
500	TB	TB	N/A

Table 4 – Runtime statistics for the Nearest Insertion algorithm

	Nearest Insertion		
# Cities	*Time* (sec)	*Path Length*	*% of Greedy*
15	0.0075966	9674.8	63.73
30	0.0309132	16081.6	81.13
60	0.0934476	22818	62.71
100	0.2228632	31068.4	85.39
200	1.398	47895	N/A
17	0.003941	12389.2	66.81
500	34.4	87914	N/A



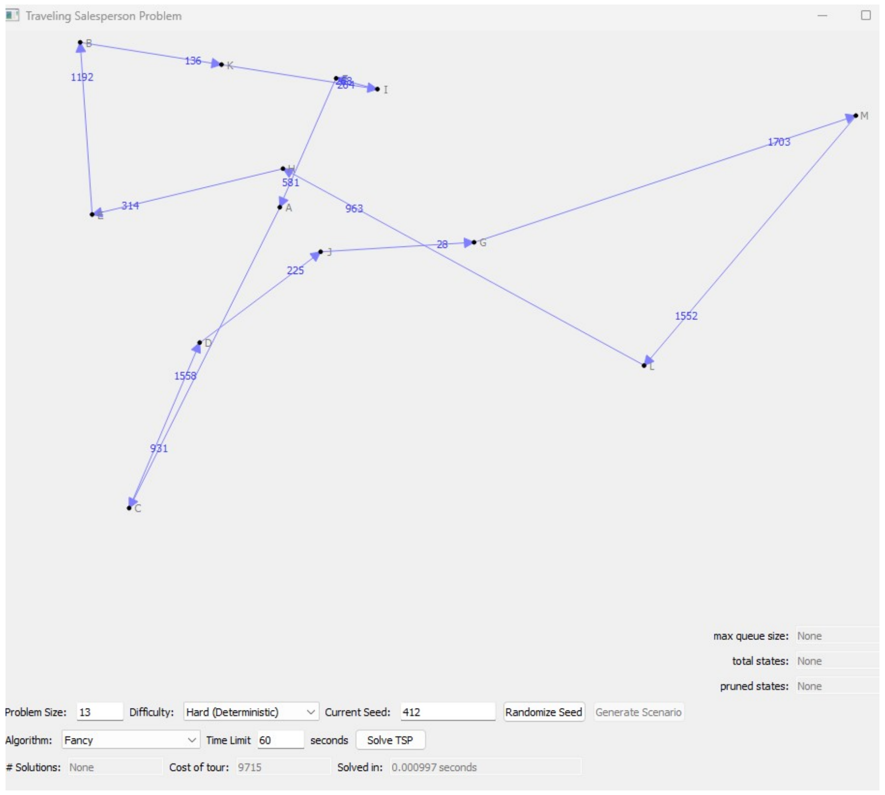
max queue size: None
total states: None
pruned states: None

Problem Size: 13 Difficulty: Hard (Deterministic) Current Seed: 412

Algorithm: Greedy Time Limit 60 seconds

Solutions: 19 Cost of tour: 11473 Solved in: 0.001026 seconds

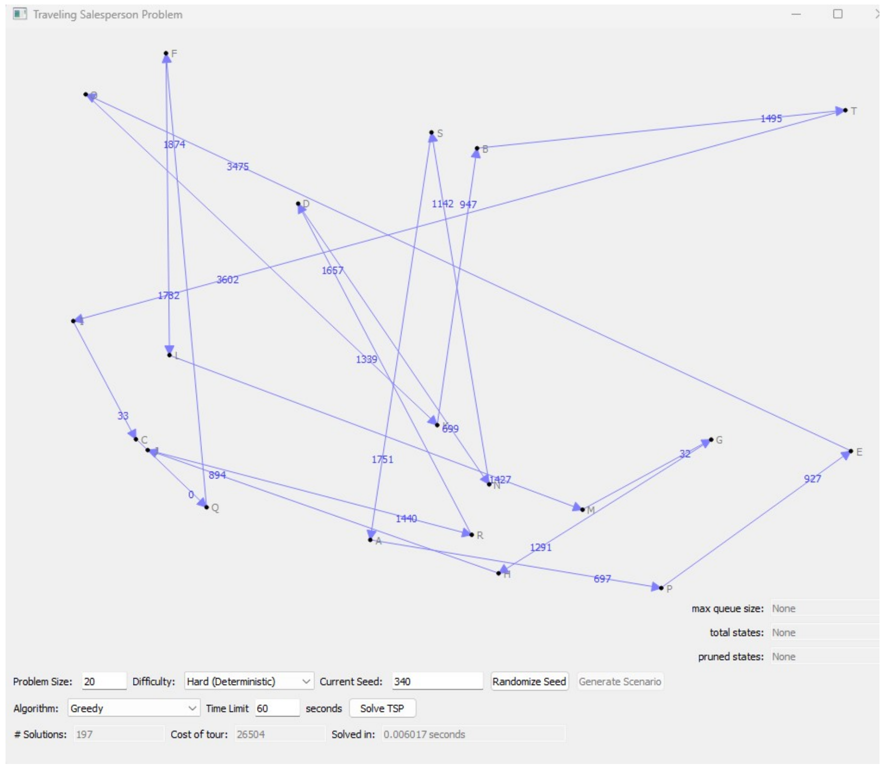
Figure 1 – The solution for the Greedy algorithm on a seed of 13



max queue size: None
total states: None
pruned states: None

Problem Size: 13 Difficulty: Hard (Deterministic) Current Seed: 412 Randomize Seed Generate Scenario
Algorithm: Fancy Time Limit 60 seconds Solve TSP
Solutions: None Cost of tour: 9715 Solved in: 0.000997 seconds

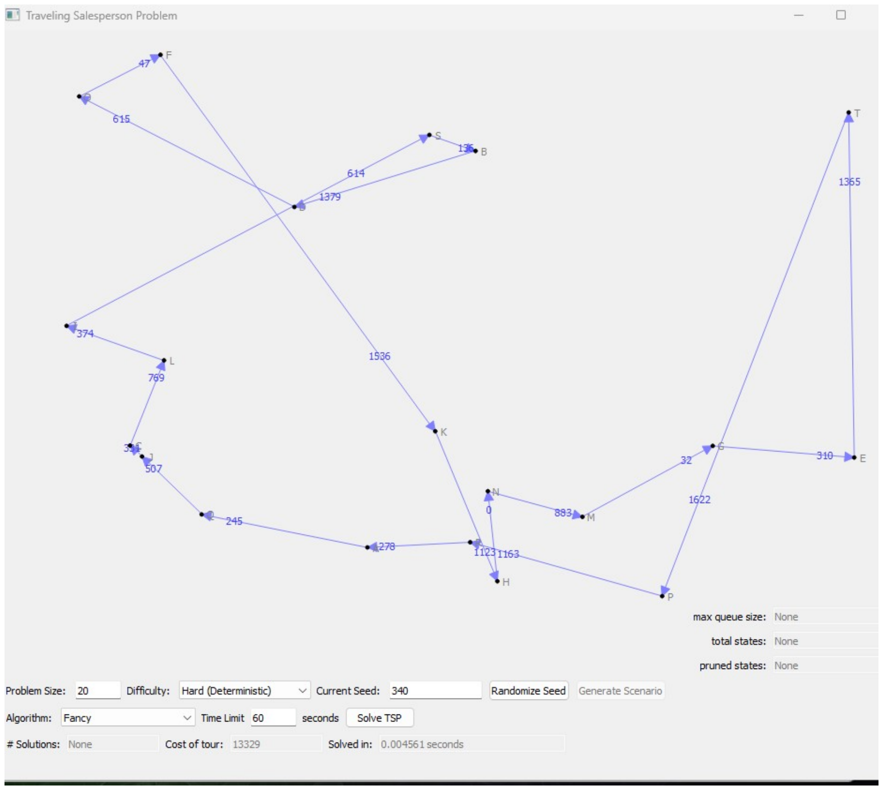
Figure 2 – The solution for the Nearest Insertion algorithm on a seed of 13



max queue size: None
total states: None
pruned states: None

Problem Size: 20 Difficulty: Hard (Deterministic) Current Seed: 340 Randomize Seed Generate Scenario
Algorithm: Greedy Time Limit 60 seconds Solve TSP
Solutions: 197 Cost of tour: 26504 Solved in: 0.006017 seconds

Figure 3 – The solution for the Greedy algorithm on a seed of 20



max queue size: None
total states: None
pruned states: None

Problem Size: 20 Difficulty: Hard (Deterministic) Current Seed: 340

Algorithm: Fancy Time Limit 60 seconds

Solutions: None Cost of tour: 13329 Solved in: 0.004561 seconds

Figure 4 – The solution for the Nearest insertion algorithm on a seed of 20

6 Future Work

With more time, we think that we can get our implementation closer to the theoretical $O(n^2)$ time complexity. In the interest of getting this algorithm to work in the time given, we had a few sections of code that were not optimized for speed but gave us a proper approximation for asymmetric graphs. After some extensive testing, we have found a bug where occasionally our nearest insertion algorithm does not include a city or two in the final solution. Getting that fixed would be great so that it can consistently deal with asymmetric graphs of any size.

References

[1] Jose Fernando Oliveira and Maria Antonia Carravilla. Heuristics and local search. https://paginas.fe.up.pt/ mac/ensino/docs/OR/HowToSolveIt/ConstructiveHeuristicsForTheTSP.pdf, 2009.