IADS Coursework 3: The Traveling Salesman Problem

Algorithm

For part C, I decided to implement a Priority Ranking Algorithm similar to that in Ali Jazayeri and Hiroki Sayama's paper "*A Polynomial-Time Deterministic Approach to the Traveling Salesperson Problem*" (1). This algorithm is based on the idea that connecting a node which is very far away from the others later on will have a huge negative consequence, and that we can use a prioritization system to decide which connections to make first. The priorities are calculated using a power function in two different steps, which considers a node's mean distance and standard deviation. The power functions use 5 hyperparameters ($\alpha, \beta, \delta, \varepsilon, \gamma$) to fine tune the results. The first power function is $p_i = \mu_i^{\alpha}\sigma_i^{\beta}$. The second power function is $c_j = \frac{\mu_j^{\delta}\sigma_j^{\varepsilon}}{d_{ij}^{\gamma}}$. In the first main step of the algorithm, each city is given a priority based on the first power function, and connected to a neighboring node with the highest priority, given that the neighboring node is not already connected to 2 other nodes. This is repeated for all cities, and guarantees that after the first step, all nodes will have at least one connected neighbor. This is why I see this algorithm as a kind of hierarchical clustering, because in each step it increases the size of the clusters, till we have a single looped tour.

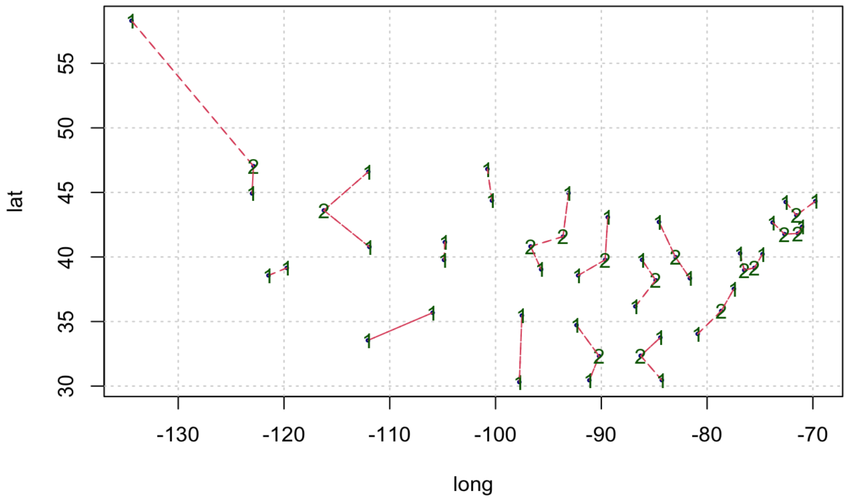

***Diagram 1*: An example of the first main step using 50 nodes. The numbers represent the degrees of each node (number of neighbors)**

As we can see in Diagram 1, the algorithm creates small clusters of 2 - 3 nodes based on their calculated priorities.

In the second main step of the algorithm, the nodes with only one neighbor are connected with their highest priority neighbor, given that the neighbor is not already connected to 2 nodes and that the new connection will not create a cycle. We check this using the isCycle() method.

Once the second main step is completed for all cities, the result is a single looped tour connecting all nodes without cycles.

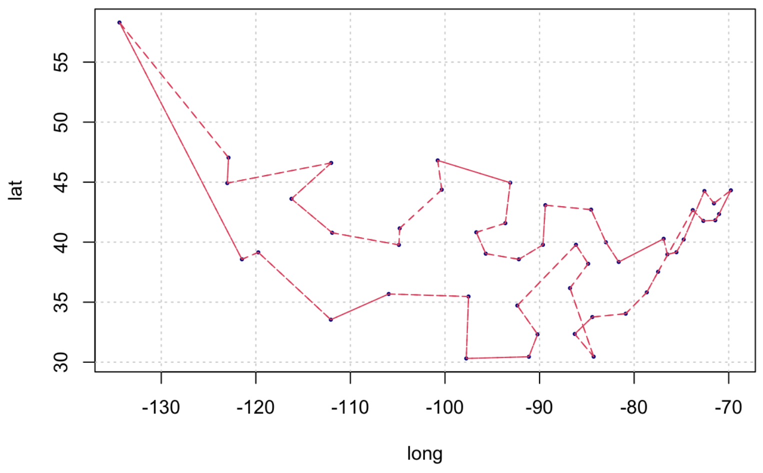

Diagram 2: **The final single looped tour connected in the second step.**

In my implementation of this algorithm, I decided to represent the existing connections as an Adjacency matrix, and almost everything else as simple lists/arrays (degrees, priorities…). Although the implementation seems complex, it simply runs this clustering algorithm for a variety of hyperparameters (α, β, δ, ε, γ). In the first and second steps, n - 1 nodes are evaluated for each node, which results in $n(n - 1)$ steps. Repeating this for a finite range of hyperparameters simply multiplies this by a constant. And so the computational complexity of the proposed algorithm is $O(n^2)$, or polynomial time, where n is the number of nodes in the graph.

References:

1. *Ali Jazayeri & Hiroki Sayama (2020) A polynomial-time deterministic approach to the travelling salesperson problem, International Journal of Parallel, Emergent and Distributed Systems, 35:4, 454-460, DOI: 10.1080/17445760.2020.1776867*

Experiments

I experimented with the different algorithms by applying each one to the same graph, recording the shortest found tour values, and comparing it to the exact solution. For larger graphs I generated ones with simple solutions like straight lines and squares. The files I use are sixnodes, twelvenodes, cities50, random_euclidean, random_metric, random_nonmetric, euclidean_straightline, and euclidean_square. The random euclidean, metric, and nonmetric contain 6 nodes each, and euclidean_straightline contains 70 nodes with length 70 in a line, hence the best distance is 140. Euclidean_square contains 100 nodes in the hape of a square, and so the perimeter should be 100. My code first generates the random graphs as files and then executes the tests. This is contained in the *tests.py* file.

File	Swap	2-Opt	Greedy	Exact	Cluster
sixnodes	9	9	8	8	13
twelvenodes	32	27	26	NA	42
cities50	8726	2872	3012	NA	11930
random_euclidean	38	37	31	31	54
random_metric	7	7	7	6	14
random_nonmetric	21	22	21	21	34
euclidean_straightline	1062	140	140	140	1860
euclidean_square	1497	140	100	100	1978

Surprisingly, my custom algorithm performs far poorer than the one suggested in the paper. As one can see, the swap heuristic, 2-opt heuristic, and greedy algorithms always find shorter paths in the test graphs.

A few noteworthy remarks for the authors of the paper for potential improvement of the algorithm:

1. The power functions used to compute the prioritization of the nodes use standard deviation as a factor. Since standard deviation is calculated using the squared distances, and the TSP costs are linear, we might consider using a linear measure such as the Mean Absolute Deviation (MAD) to make for a better prioritization measure.
2. Of course, this algorithm uses Euclidean distances, but if we were using real latitude and longitude coordinates, we would need to use geodesic distance.
3. Lastly, the algorithm could be more efficient if there were fewer hyperparameters to search for. Currently the functions use $\alpha, \beta, \delta, \varepsilon, \gamma$ in the equations $p_i = \mu_i^{\alpha}\sigma_i^{\beta}$ and $c_j = \frac{\mu_j^{\delta}\sigma_j^{\varepsilon}}{d_{ij}^{\gamma}}$. Because power functions are strictly increasing functions, the first equation can be modified to use only one hyperparameter, and the second can be modified to only use two. Reducing the number of hyperparameters from 5 to 3 will significantly decrease the overhead cost of running the algorithm. The new equations would look something like $p_i = (\mu_i\sigma_i)^{\beta}$ and $c_j = \frac{(\mu_j\sigma_j)^{\varepsilon}}{d_{ij}^{\gamma}}$, with hyperparameters $(\beta, \varepsilon, \gamma)$.

The paper’s findings seem to differ from mine, and this could be explained by differences in implementation and differences in the test graphs.