IADS Coursework 3 Report
Heuristics for the Travelling Salesman Problem

My custom algorithm 'Temperate'

I formulated my own custom algorithm for finding approximate optimal routes in polynomial time.

How it works

I wanted to create an algorithm that finds optimal paths by prioritizing the cities/nodes which have the highest mean distances from other cities. This prioritization works by creating fragments/edges (represents the transition between two cities/nodes) between the city/node with the highest mean distance and expanding it with its best possible path (shortest). This is iterated for all cities from highest to lowest mean distance and placed appropriately in a priority queue (descending order of mean values) until all nodes are used up. Once a priority queue of fragments/edges is found then the route can begin to be created. The route is initialized with the first fragment in the priority queue. Thereafter node expansion is based upon availability in the priority queue or best possible transition from the given node. By this I mean that based upon the given the node we need to expand (last node in current route), the algorithm first checks if this node can be found in a fragment in the priority queue. If so, the nodes for this given fragment are added to the route. Otherwise, the algorithm finds the best available transition for the given city/node and adds it to the route. This process is continued until a complete route is formed.

In order to ensure consistency throughout this process lists are used to represent all nodes/fragments that have already been explored.

Overall thoughts

I realized upon implementation it almost works in the opposite way to the greedy algorithm in which rather than prioritizing using the shortest distances possible, mine prioritizes **not** using the longest distances possible, this is why I decided to name this algorithm 'Temperate'(the antonym of greedy). Overall, I thought this concept would be effective as upon this prioritization, the most significantly large distances/costs can be reduced to their minimum possible value.

Results

Generally my results were on the same par as the Greedy algorithm (on average a 3,6% higher tourValue), which I was happy with given the similarities and disparities between their associated prioritization techniques. However, what I found particularly exciting was the results of my algorithm after being parsed by the refinement algorithms Swap and 2-Opt. The results upon parsing the result of my algorithm with the combination of these refinement algorithms produced the lowest cost routes throughout all given implementations.

tourValue OUTPUTS												
File	**Identity**	**Swap**	**2-Opt**	Swap & 2-Opt	**Greedy**	w/ Swap	w/ 2-Opt	w/ Swap & 2-Opt	**Custom algorithm**	w/ Swap	w/ 2-Opt	w/ Swap & 2-Opt
cities25	6489	5027	2211	2233	2587	2535	2196	2196	2573	2555	1993	2180
cities50	11842	8707	2781	2686	3011	2921	2687	2654	3278	3190	2575	2575
cities75	18075	13126	3354	3291	3412	3304	3160	3160	3494	3450	3155	3095
six nodes	9	9	9	9	8	8	8	8	9	9	8	8

w/ Swap & 2-Opt: denotes that after the given algorithm is executed, the result is refined through the implementation of the swapHeuristic and TwoOptHeuristic functions. These functions are executed in the order expressed above (custom -> swap -> 2-opt).

Green boxes used to represent lowest cost output for the given file.

Experiments

I formulated my own functions that generate random Euclidean/Metric graphs in order to test and compare the efficiency of different algorithms at a large scale. (All code within tests.py)

Generating random graphs: *createRandomMetricGraph(), createRandomEuclideanGraph()*
In order to generate random graphs that could be used as input to any of my algorithms I decided to generate them individually in the form of a text file. The given graph size to generate is specified by the size input, and the cost of edges for a given graph was calculated by choosing a random number between the upper and lower bound inputs.

Comparing efficiency of different algorithms using random graphs: *calculateCostDiffs()*
In order to compare the efficiency of different algorithms we can compare how well they do in calculating the tourValue for a given graph. To make this comparison accurate and reliable we must do this at a large scale by comparing these algorithms multiple times with different graphs. So, to achieve this, I created a function which calculates the average tourValues for all the different algorithms over **n** tests (n = the input value for the desired number of tests). In each test, the tourValue is calculated for all algorithms against two randomly generated graphs of specified size (a Euclidean graph of input size **x**, and a Metric graph of input size **y**).

In order to ensure high accuracy and reliability, all outputs below were produced using the average tourValue produced over 500 random tests for Euclidean/Metric graphs of the specified size.

Average tourValue OUTPUTS for 500 tests													
Graph type	**# Nodes**	**Identity**	**Swap**	**2-Opt**	Swap & 2-Opt	**Greedy**	w/ Swap	w/ 2-Opt	w/ Swap & 2-Opt	**Temperate (custom)**	w/ Swap	w/ 2-Opt	w/ Swap & 2-Opt
Euclidean	25	3146	2434	1058	1058	1181	1161	1030	1030	1168	1136	1020	1020
	50	12730	9894	2993	2993	3386	3335	2913	2913	3360	3291	2890	2890
	75	28940	22424	5493	5493	6217	6127	5336	5336	6216	6096	5300	5300
Metric	4	7	6	6	6	6	6	6	6	6	6	6	6
	6	18	13	12	12	12	12	11	11	12	12	12	12
	8	27	19	16	16	16	16	15	15	17	16	15	15

RESULTS:

As evident by all the green marked cells the overall most efficient algorithm is:

Temperate (custom) w/ Swap & 2-Opt

I am very happy with this result as it validates the reliability of my results obtained in Part C, and ultimately the efficiency of my own custom algorithm.

Without including the combined implementation of any algorithms, the overall algorithm efficiency rankings are:
(all tourValues are taken as the average % differences against the top algorithm)

For Euclidean graphs

1. **2-Opt**
2. **Temperate (custom) + 11.94% tourValue**
3. **Greedy + 12.65% tourValue**
4. **Swap + 222.95% tourValue**

For Metric graphs

1. **Greedy & 2-Opt**
2. **Temperate (custom) + 2.08% tourValue**
3. **Swap + 9.03% tourValue**

So overall, according to the trends in the result data, I have found that:

1. Temperate w/ Swap & 2-Opt is the overall most efficient algorithm for Euclidean graphs.
2. Greedy w/ Swap & 2-Opt is the overall most efficient algorithm for Metric graphs.
3. 2-Opt is the overall most efficient raw (implemented independently) algorithm for both Euclidean and Metric graphs.