IADS coursework 3

Part C:

The algorithm I chose to implement is an already existing approximation algorithm for the TSP. It uses the idea of minimum spanning trees in order to find a traversal path in a weighted graph that is close to the optimal one. A minimum spanning tree is a subset of the edges of a weighted graph, such that all the vertices of the graph are connected to each other, there are no cycles, and we have the minimum possible total edge weight. Therefore, it's easy to see why this would be a good approximation for the optimal solution of the TSP.

The main idea of the algorithm is:

1. Find minimum spanning tree of given graph with starting node 0.
2. Perform DFS on it and return the list with the nodes in the order they were visited, without including duplicates.

For step 1 I used Prim's algorithm, which finds a minimum spanning tree with a particular starting node for a graph. This algorithm was first discovered by Vojtech Jarnik in 1930 and was later republished by Robert Prim and Edsger Dijkstra in the 1950's. The way that it works is:

1. Create an empty set that will keep track of vertices already included in the minimum spanning tree.
2. Assign a key value equal to infinity to all vertices in the graph, except for the starting vertex, which will have a key value of 0.
3. Pick the vertex from the graph with the smallest key value which is not already in the set (let's call it V).
4. Add V to the set.
5. Iterate through all of V's adjacent vertices. If the weight (w) from an adjacent vertex of V to V is smaller than the adjacent vertex's previous key value, then update its key value to w.
6. Repeat steps 3 to 5 until the set includes all the vertices of our input graph.

The algorithm returns a minimum spanning tree represented as a list, where the element and each index i shows the parent of the i^{th} node in the tree.

The complexity of Prim's algorithm is O(n^2), where n is the number of nodes of our graph, since there is an outer loop that repeats until all nodes are included in the minimum spanning tree, and an inner loop that iterates through all adjacent nodes of each selected node (in the worst case, a node will be adjacent to all other nodes in the graph).

After creating the minimum spanning tree, we need to visit all of its nodes to come up with a path. For this purpose, I used DFS (depth first search), first investigated in the 19th century by Charles Pierre Tremaux. It starts at the root of a tree and explores as far as possible along each branch (until it reaches a leaf) before visiting a new branch. It returns the list of nodes in the graph in the order they were visited. Given that we have thoroughly examined this algorithm in this course, I will not go into more detail as to how it works. The only difference in my implementation is that I had to adjust it in order to work with the specific format in which the minimum spanning tree is represented after running Prim's algorithm. That's why I added the argument `neighbors` to keep track of the adjacent vertices of each node.

The complexity of DFS is O(n + e), where n is the number of nodes and e the number of edges in our graph.

Putting it all together, we see that the complexity of the approximation algorithm used is O(n^2 + (n + e)) = O(n^2 + e), which is polynomial time.

Part D:

For my experiments I wrote two functions to generate random graphs, the `generateEuclideanGraph()`, which generates a file in the format of `cities50` and represents a graph for the Euclidean TSP, and the `generateGeneralGraph()`, which generates a file in the format of `twelvenodes` and represents all the general cases of the TSP.

Both functions take as argument the number of nodes `numOfNodes` that the graph should have. In `generateEuclideanGraph()` we also have the parameters `widthOfPlane` and `heightOfPlane` which let us define the maximum x and y coordinates respectively that our point-nodes should have. On the other hand, in `generateGeneralGraph()` we have a parameter `maxDistance` which allows us to define what is the maximum distance that we want to have between two point-nodes.

I have also implemented the function `bruteForce()` , which returns the minimum cost of a specific graph by finding the cost of all possible permutations and returning the smallest of those. This function is useful for finding the actual minimum cost and comparing it with the result that we get from the different heuristics that we have implemented in other tasks. This way we can determine which heuristic consistently returns the cost that is closest to the true minimum value. This is to be used for small graphs, with up to 10 nodes, since it has complexity O(n!) and it becomes really slow with graphs with over 10 nodes.

The tables below show some of the results that the testing algorithms gave on some random general graphs:

Minimum Cost with bruteForce()	Cost After swap and TwoOpt	Cost after Greedy()	Cost after myAlgorithm()
103	103	125	103
199	205	235	217
46	46	46	46
118	118	118	118
95	95	95	95
26	26	50	26
27	27	27	29
11	11	11	11
113	113	128	151
70	70	70	156

Table 1: Results regarding 10 random general graphs with 5 nodes

Minimum Cost with bruteForce()	Cost After swap and TwoOpt	Cost after Greedy()	Cost after myAlgorithm()
203	215	259	292
77	77	83	125
192	192	244	226
153	153	177	205
21	27	21	40
39	39	45	54
25	27	35	38
223	223	239	282
32	38	32	43
150	150	202	307

Table 2: Results regarding 10 random general graphs with 8 nodes

Minimum Cost with bruteForce()	Cost After swap and TwoOpt	Cost after Greedy()	Cost after myAlgorithm()
104	113	133	165
76	91	101	136
184	185	250	268
110	133	166	238
143	155	144	212
121	131	153	242
55	74	92	96
82	86	107	124
233	252	307	330
88	88	117	154

Table 3: Results regarding 10 random general graphs with 10 nodes

By examining the tables above, we can see that in the case of the 5-node graphs, the cost returned by the different heuristics and approximation algorithms are generally very close to the actual minimum cost. There are a lot of cases where all or most of the algorithms return the optimal value. The combination of the swap and the TwoOpt heuristic seems to be the closest approximation, since in 90% of cases it returns the actual minimum cost. The greedy algorithm the algorithm implemented in part C both only return the actual minimum cost 60% of the time, however Greedy() consistently returns values that are closer to the optimal one, so it's a better approximation than myAlgorithm().

As we examine graphs with more nodes, we have similar observations to the ones above. The difference now is that the value returned by each algorithm deviates more on average from the optimal cost (hence the cases where any algorithm returns exactly the minimum cost are far fewer); however the combination of swap and TwoOpt still has the smallest deviation, with Greedy() and myAlgorithm() following, in this order.

Now let's look at the results returned by the tests when performed on some random Euclidean graphs:

Minimum Cost with bruteForce()	**Cost After swap and TwoOpt**	**Cost after Greedy()**	**Cost after myAlgorithm()**
64.286	64.286	64.286	64.286
135.824	135.824	142.041	142.041
159.300	159.300	159.300	159.300
140.543	140.543	140.543	142.024
128.227	128.777	128.777	128.777
73.828	73.828	75.801	75.801
209.763	209.763	209.763	209.763
89.110	89.110	99.684	99.684
152.913	152.913	163.276	174.307
89.209	89.209	89.209	89.209

Table 4: Results regarding 10 random Euclidean graphs with 5 nodes

Minimum Cost with bruteForce()	**Cost After swap and TwoOpt**	**Cost after Greedy()**	**Cost after myAlgorithm()**
109.686	109.686	124.374	114.936
171.423	171.423	217.240	196.537
263.108	263.108	269.784	269.784
151.033	151.033	159.738	159.738
210.033	210.033	231.921	216.758
103.760	103.760	108.745	108.745

152.694	152.694	154.004	175.778
251.318	251.318	306.718	282.193
136.393	136.393	157.810	145.070
129.589	129.589	149.162	147.922

Table 5: Results regarding 10 random Euclidean graphs with 8 nodes

Minimum Cost with bruteForce()	Cost After swap and TwoOpt	Cost after Greedy()	Cost after myAlgorithm()
172.330	172.330	172.330	228.730
187.293	187.293	247.852	218.100
143.247	143.247	153.855	169.010
91.667	91.667	105.228	108.939
85.593	86.068	92.665	110.896
209.564	209.564	226.097	213.755
273.299	279.020	315.143	316.407
86.345	86.345	96.930	94.019
224.142	224.142	238.733	309.010
119.220	120.170	126.908	126.908

Table 6: Results regarding 10 random Euclidean graphs with 10 nodes

In the Euclidean case we notice something a little different. First of all, regardless of the number of nodes in the graphs, in almost 100% of cases the combination of swap and TwoOpt returned exactly the optimal cost, making it by far the best approximation. We also notice that the deviation of the results of Greedy() and myAlgorithm() from the optimal value did not increase significantly as the number of nodes increased, unlike the case of the general graphs. Therefore, performance doesn't seem to be so much affected by the number of nodes.

Nevertheless, the most interesting observation is that in this case myAlgorithm() performs better on average than Greedy(), since in the majority of cases its result is closer to the actual value. That is because the Euclidean case falls under the metric case, where distances between nodes follow the triangle inequality (i.e. it's always cheaper to go from A to B rather than from A to C and from C to B). In myAlgorithm() the minimum path returned is the order in which the nodes of the minimum spanning tree were visited by DFS. However, in that list we did not include duplicate nodes. That means that if we have a tree like this:

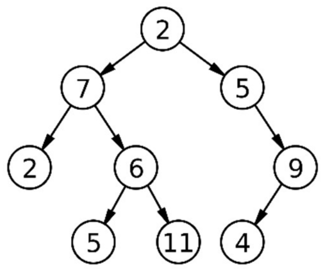

Image source: https://en.wikipedia.org/wiki/Binary_tree

then the path returned will be [2,7,2,6,5,11,5,9,4] and not [2,7,2,7,6,5,6,11,6,7,2,5,9,4]. In the general case the second path might have been more cost effective, since it's not guaranteed that going directly from one point to the next is cheaper than going through other nodes. However, since this is indeed guaranteed in the Euclidean case, the cost of the first path will always be closer to the actual minimum cost. Hence, it's natural that myAlgorithm() performs better in the Euclidean case than in the general case.

The only problem with the method used above is that it cannot be used for graphs with more than 10 nodes, since the brute force algorithm becomes extremely inefficient and we cannot get an exact result for the minimum cost. For this purpose, I have written another function, generateLargeGraph(), which generates a Euclidean graph where all the points are in a straight line (i.e. have the same y coordinate), with the left-most node being node 0 (i.e. it has the smallest x coordinate), followed by node 1 and then 2 etc (see figure 1 below). This way we know that the minimum cost will always be given when running tourValue() on the identity permutation of our graph.

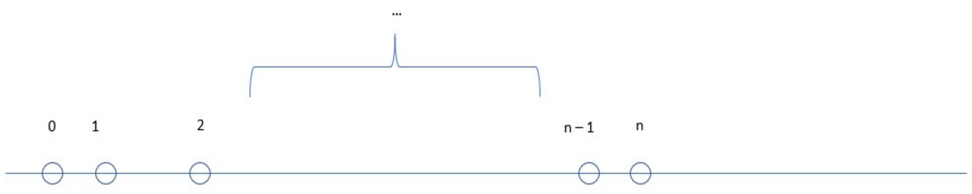

Figure 1: example graph generated by generateLargeGraph() with n nodes

In this case, however, all of the approximation algorithms don't get to do much work, since the minimum cost route is always the identity permutation, which is the one that our graph is initialised with, and so all of them always return the correct value for the minimum cost. The ideal situation would be if we could generate a graph where all the nodes are in a straight line, with zero always on the far left, but not in the order 0,1,2…,n. Instead they should be in another (fixed) order p, so that we would know that if we run tourValue() on p we would always get the minimum cost.