Informatics 2A: IADS Coursework 3 Report

Algorithm

My algorithm is a version of Nearest Insertion which is a typical heuristic used in the Travelling Salesman Problem. The main idea of the algorithm is that, starting at 0, it looks for the shortest possible distance from an unvisited node to any of the nodes in the current sub-tour. Not just the 'leading node' (current end-node of the sub-tour) like most greedy algorithms.

For example if you have a current sub-tour of 0->1->2 and say you have a list of unvisited nodes [3,4,5] this algorithm may search through the distances of all the nodes in the unvisited list and find that 1->4 has the shortest distance of all the nodes in the sub-tour to all of the nodes in the unvisited list . So if this is the case then 4 is inserted after 1 as so 0->1->4->2.
This continues; say 5 has the shortest distance of a node in all possible nodes in the sub-tour to a node in the unvisited list and that shortest distance is node 0 to node 5 then 5 is inserted after 0.
So 0->5->1->4->2
And so on.

Pseudocode

```
1.create a new Tour list with starting node x
2.create a visited node list with starting node x in it as well
3.loop from 0 to (number of nodes in graph -1)
4.call shortestDistance function
5.insert into list at node y's index +1 the node x with the shortest distance to index y  found via the
shortestDistance function
6.end loop
7. Return the new Tour list which should have a completed tour in order

shortestDistance function
1.notChosenNodes = (all possible nodes) - visitedNodes
2.take currentLowestIndices (i.e tuple of some node x and some node y who have the lowest distance) as last
node of the current sub-tour (i.e tail of newTour list) and head of the notChosenNodes list
3.loop for each node I  in the current sub-tour
4.loop for each node j  in the notChosenNodes list
5.if cost(i,j) is less than cost(currentLowest,j) and i != j and j is not is visitedNodes
6. If so then set currentLowest to j  and currentLowestIndices to (i,j)
7.end loop
8.end loop
9. Return tuple currentLowestIndices to main function which now has the two nodes (with node I currently in the
sub-tour) with the shortest distance between them.
```

Pros and Cons of the algorithm:

Pros
-This algorithm could be looked at as a slight variation or modification of a Greedy algorithm with instead of finding the node with the shortest distance from the tail of the current sub tour it instead looks at the shortest distance of any node in the current sub-tour. This means that a tour under this heuristic will have some of the shortest distances possible between nodes in its tour. Furthermore unlike some Greedy algorithms this heuristic does not eliminate some better possible options earlier. For example if connecting node x to node y is the shortest distance for x under a greedy algorithm but a connection of say w to y would have lead to an overall shorter tour this heuristic can accommodate for that and insert w e.g x->w->y.

-Another pro of this heuristic is that unlike say 2-opt or the swap heuristic which need an already decided underlying tour to work, the Nearest Insertion can build a tour from scratch. This is beneficial in cases where a solution may not be known or where it is difficult to write up an initial tour because of the number of nodes/ complexity of the graph etc.

Cons
-Unfortunately this heuristic comes into some of the same issues that other Greedy algorithms have. For one this algorithm cannot find the optimal solution each time for every graph and in some cases the solution returned could be the unique worst possible solution. This is because while this heuristic can decide on the shortest distance from a node to any node in the current sub-tour it is still only basing its decision on information at one step and not how this insertion may affect the whole sub tour.
-Furthermore this algorithm is slower than say SwapHeuristic and Greedy since it has a worst case time complexity of O(n^3) since it has to search through and compare all the possible distances in the sub tour and not just all the possible distances of the last or tail element in the sub-tour like Greedy.

Time Complexity of Algorithm

Nearest Insertion runs in O(n^2) time making it a polynomial time algorithm. This is easily shown from the pseudocode; we are looping n times in total to add n nodes needed for tour. Then for each of those n nodes we are looping a maximum of n times to find the shortest possible distance from that node to an as yet unvisited node. Therefore n*n = n^2.
However this is obviously an upper bound as for each of the n nodes we are never searching through the full list of n nodes to find the one with the shortest distance; we are always searching through the unviisted nodes i.e n - x nodes.

Experiments

It should be noted for all the experiments graphs were randomly generated, within certain parameters, using the code found in tests.py. These randomly generated graphs were then run with the functions through Terminal and the time taken (in seconds) for each algorithm was found using timeit. The numbers below are averages since the experiment was repeated 4 times for each different sets of parameters using a different randomly generated graph each time.

Euclidean Distance

	SwapHeuristic	**TwoOptHeuristic**	**Greedy**	**NearestInsertion**
Euclid- <=100nodes <=1000 coords	0.00069266458740458	0.31216273724567	0.0028443454939	0.025404904736206
Euclid- <=500nodes <=500 coords	0.00616684270789	95.220181785757	0.2459963192814	4.070587462018
Euclid- <=100nodes <=10000 coords	0.00063893847982	0.07430405524792	0.00117596698692	0.00808609274099
Euclid- <=500nodes <=10000 coords	0.00779010634869	117.99842170071	0.30245198305541	4.9442074523152

For this series of experiments I looked at what happens when the number of nodes in the graph is increased as well as the range of possible x and y coordinate values and the affect these have on the times of the algorithms.

What is interesting to note however is that when the number of possible nodes in a tour is multiplied by 5 the average time of the algorithms goes up almost by a factor of 100 in some cases. This is particularly clear with the TwoOptHeuristic, which is the slowest algorithm in general, where the algorithm becomes very slow very quickly with just a linear increase in the number of nodes. This makes trying to plot the time difference between two opt and other heuristics very difficult because of the such extreme difference in magnitude.

Furthermore the increase in the range of the coordinates has no real effect on performance of the algorithms. Having tried with 1000 vs 10,000 with the table above I tried a more drastic difference in magnitude to see if there was any difference in time:

	SwapHeuristic	TwoOptHeuristic	Greedy
<=1000 coords	1.5581026673317E-06	1.27917155623436E-06	1.32201239466667E-06
<=10x10^16 coords	1.93319283425808E-06	1.07311643660069E-06	7.19912350177765E-07

As few can tell from the values, increasing the coordinate range makes no difference to the time overall with it being slower for SwapHeuristic but faster for TwoOptHeuristic and Greedy. However what's interesting is that my SwapHeuristic is the only algorithm to use arithmetic in it with the other two simply use comparisons when dealing with the distances in the graph. It may be this arithmetic that is reason why SwapHeuristic is now slightly slower.

Planted Solution

For this I created a function in tests that generates a graph that makes a square the idea being that by following the perimeter of the square we reach the shortest or optimal solution. It should be noted that for this solution I had to randomise self.perm before running the algorithms as otherwise the optimal solution would already be there.
SwapHeuristic was unable to find such an optimal solution, returning the worst result of the 4 algorithms with a tour cost of 62.6485. for a tour of 20 nodes. This took a relatively short time of 0.000101.. It should also be noted here that I imagine the randomness of the initial self.perm will have a small value in deciding the tour value. If the random self.perm is close to a possible optimal solution (e.g 0, 1,3,2….19) then by swapping the values you could reach a value close to the actual optimal value. However if the random self.perm is not close at all to a possible optimal solution i.e no ordering of the node (7,18,3,12…) then a poor tour value will be returned.

TwoOpt was able to find an optimal solution going around the perimeter of the square and returning the best case tour value of 20. Greedy performed the best with this experiment surprisingly returning an optimal solution in a time of 9.9638..x10^-5. This continues when the number of nodes goes up; it takes TwoOpt 2.583622… seconds and it takes Greedy only 0.03984… seconds.

However this success of Greedy seems to be only the case for the optimal solution being a square graph. When the optimal graph is changed to a rectangular shape Greedy fails and returns a worse possible solution than TwoOpt as the table shows:

	Tour Returned	Time Taken
SwapHeuristic (Square soln, 20 nodes)	62.6485582929001	0.000101907178759575
TwoOptHeuristic (Square soln, 20 nodes)	20.0	0.00369907799176872
Greedy (Square soln, 20 nodes)	20.0	0.000102924881502986

	Tour Returned	Time Taken
SwapHeuristic (Square soln, 200 nodes)	5337.84691139469	0.000649713911116123
TwoOptHeuristic (Square soln, 200 nodes)	200.0	2.58362669800408
Greedy (Square soln, 200 nodes)	200.0	0.039846827974543
SwapHeuristic (Rectangle soln, 20 nodes)	57.5253552946224	0.000111764064058661
TwoOptHeuristic (Rectangle soln, 20 nodes)	22.8284271247462	0.00519240507856011
Greedy (Rectangle soln, 20 nodes)	34.9137144251435	0.000163306947797537

I believe this is because when you get to a corner in the rectangle solution in Greedy the algorithm literally cuts corners and finds a value closer to it not realising it is missing the nodes at the corner which it will have to come back to and hence increase the overall tour value.
We can clearly see this with the self.perm solution of Greedy for 20 nodes (the rectangle in this case has length 6 breadth 4)

```
[12, 11, 10, 9, 8, 5, 7, 4, 3, 2, 1, 0, 18, 17, 16, 15, 14, 13, 19, 6]
```

It starts off going up a side of the rectangle with 12->11->10->9->8 but when it hits the corner it instead chooses 5 being closer and then has to backtrack back to the 7 after the 5 and then at the very end has to backtrack to the 6 as well hence the increase in the cost of the tour.