Algorithm

For my own algorithm I considered two main types of algorithms which may help solve the problem: Algorithms which construct a new tour given a certain heuristic(Tour Construction) and stop once a solution is found and algorithms which try to optimise(Tour Improvement) an already existing tour given certain heuristics.

Ultimately I decided to write a tour construction algorithm as I wanted to see how effective a different tour construction algorithm would be compared to for example TwoOpt which generally gives very decent results.

For my own algorithm I decided to use insertion heuristics as a means of reducing overall tour cost. The basics of insertion heuristics are to start with a tour consisting either of a subset of all nodes(cities) or of a single edge. Then one by one insert each remaining unvisited city into the tour by the according insertion heuristic, until all the cities have been visited. Once all the cities have been visited, the insertion algorithm will have constructed a complete tour.

There are many variants of insertion heuristics, such as cheapest insertion, convex hull insertion, farthest insertion, nearest insertion, etc. The one I decided to try and implement was the nearest insertion heuristic.

The basic idea of nearest insertion is start by making an edge of two nodes which are closest to each other and make them the initial tour. Then select a node still not in the tour that has the shortest distance to any of the nodes in the tour. Now find the edge in the tour which minimises the cost of inserting the selected node in between that edge. The cost is given by cik + ckj - cij (where cij is the distance between node i and j). Once the edge has been found insert the selected node in between that edge. Keep repeating this for all the nodes still not in the tour until every node is in the tour.

My implementation of the nearest insertion algorithm starts by making a list of all the nodes available(nodesLeft). It then takes node 0 and adds it to the tour and removes it from the list of available nodes. Note: the algorithm can start from any node, but for simplicity I implemented mine to start at node 0.
It then finds which node from nodesLeft is closest to the single node in the tour and adds it to the tour, removing it from nodesLeft. Now that there are two nodes in the tour we have an edge i,j.
Now the algorithm will go through every node in nodesLeft and call it k. For each k it will check all the values i,j in the tour and if the values i,j are next to each other i.e. they form an edge it will calculate cik + ckj - cij for that edge, if this is lower than the previous lowest dist (initially initialised to an arbitrarily high value). This then becomes the lowest value, and the indexes of i,j are saved, so that we know where to insert k if this turns out to be the best value.
The algorithm keeps looping through the value i,j, until it finds which edge i,j which minimises cik + ckj - cij and inserts k in between that edge and removes it from nodesLeft.
The algorithm then repeats this process until there are no more nodes in nodesLeft, meaning that a complete tour has been constructed.

Algorithm

It is very easy to modify my algorithm to a farthest insertion algorithm. All that would need to change was we would pick the edge that maximises cik + ckj - cij instead of the one that minises it. After doing some very brief testing using farthest insertion, I found that the value tour value it gives, was slightly higher than nearest insertion, so I stuck with the nearest insertion heuristic.

Experiment

For my experiments I began by writing code to generate some graphs. I then adjusted various parameters to see how they would affect the algorithms in terms of their runtime and the optimality of the solution they found.
I started by looking at the Euclidean case. The first thing I tried was varying the number of nodes/cities. I found that generating 15 files was the optimal for getting reliable results but not taking forever to run.
For this experiment the range of nodes was kept at 0 to 100 inclusive.

My findings can be seen in the table:

Varying number of nodes		Number of files = 15		
	Swap	2-Opt	Greedy	Nearest Insertion
N = 10	Improvement = 23.32%, time = 0.0038s	Improvement = 46.37%, time = 0.0075s	Improvement = 42.09%, time = 0.0012s	Improvement = 19.84%, time = 0.0026s
N = 25	Improvement = 22.23%, time = 0.02s	Improvement = 65.99%, time = 0.11s	Improvement = 59.95%, time = 0.0047s	Improvement = 42.29%, time = 0.027s
N = 50	Improvement = 22.98%, time = 0.073s	Improvement = 76.5, time = 1.06s	Improvement = 72.38%, time = 0.019s	Improvement = 59.44%, time = 0.18s
N = 100	Improvement = 21.71%, time = 0.22s	Improvement = 83.29, time = 8.63s	Improvement = 80.7%, time = 0.08s	Improvement = 70.91%, time = 1.4s
N = 150	Improvement = 22.05%, time = 0.46s	Improvement = 86.73, time = 33s	Improvement = 85.06%, time = 0.2s	Improvement = 75.66%, time = 4.64s

Looking at things algorithm by algorithm, it is clear that for the swap algorithm varying the number of nodes had no effect on how effective the tour cost reduction was, in each case it was around ~22% and its runtime seemed to grow steadily with the number of nodes.

With 2-Opt it seemed that the more nodes in the file, the better the improvement in tour cost, however the time for 2-Opt to run grew exponentially to the number of nodes, so although it returned a very nice result, it was the slowest algorithm of the 4.

Similarly to 2-Opt the greedy algorithm was more effective, the higher the number of nodes was. However the tour it returned was not quite as optimal as 2-Opt, but it's runtime was drastically better.

My own algorithm, which is a nearest insertion algorithm, gave very interesting results, as it too became better with more nodes. However, for a very low number of nodes, my algorithm gave very poor results, worse than even swap. It was also not very fast, although it was faster than 2-Opt, but gave much less optimal results.

Algorithm

Weighing up the time taken and optimality of results, I would say the best algorithm, if there are many nodes/cities, is the Greedy algorithm, as it almost matched 2-Opt for optimality but was hugely faster. If there are less nodes or if time is not a factor then 2-Opt would be best.

Next I considered varying the window of the plane, to see how that would affect results. I kept the number of nodes at 50 for all the experiments below. These were my findings.

Varying window of plane		N = 50		
	Swap	2-Opt	Greedy	Nearest Insertion
Range = (0.5)	Improvement = 25.13%, time = 0.058s	Improvement = 79.73%, time = 0.83	Improvement = 76.5%, time = 0.023s	Improvement = 58.23%, time = 0.2s
Range = (0.25)	Improvement = 22.25%, time = 0.061s	Improvement = 76.28%, time = 0.93s	Improvement = 73.71%, time = 0.019s	Improvement = 63.69%, time = 0.18s
Range = (0.100)	Improvement = 21.82%, time = 0.057s	Improvement = 76.65%, time = 1.02s	Improvement = 73.65%, time = 0.02s	Improvement = 61.72%, time = 0.19s
Range = (0.200)	Improvement = 24.14%, time = 0.056s	Improvement = 76.66%, time = 0.99s	Improvement = 73.62%, time = 0.022s	Improvement = 59.65%, time = 0.23s

It is clear that for these algorithms varying the window of the plane has no effect on either runtime or on the optimality of the solution found.

Next I did a similar test for some non metric graphs. First by once again varying the number of nodes:

Varying number of nodes		Weight range = (0,50)		
	Swap	2-Opt	Greedy	Nearest Insertion
N = 5	Improvement = 21.15%, time = 0.018s	Improvement = 67.47%, time = 0.12s	Improvement = 63.35%, time = 0.005s	Improvement = 44.57%, time = 0.028s
N = 10	Improvement = 21.93%, time = 0.22s	Improvement = 84.26%, time = 7.7s	Improvement = 81.55%, time = 0.1s	Improvement = 70.3% time = 1.37s
N = 15	Improvement = 22.4%, time = 1.1s	Improvement = 89.88%, time = 1m 41s	Improvement = 88.21%, time = 0.83s	Improvement = 78.29%, time = 16.8s

Like in the euclidean case this had similar effects on the algorithms. The swap algorithm once again showed no change in optimality regardless of the number of nodes, staying at a constant level of ~22%, and runtime grew steadily.
The 2-Opt algorithm was once again most effective in reducing the tour cost, but took incredibly long. So long that I could only test very low numbers of nodes, because 2-Opt was the limiting factor.

Greedy also became more optimal with more nodes, and it was also the fastest algorithm.
Nearest Insertion followed it's trend of being in between all the algorithms, running slower than greedy/swap but returning results much better than swap but now quite as good as greedy/2-Opt.

I then tried to vary the range of the weights, however this also proved to have no effect on any of the algorithms in terms of runtime and optimality, due to the extremely long runtime of 2-Opt for higher numbers of nodes, I kept the number of nodes at 10 for the below experiment:

Algorithm

Varying weight range		N = 10		
	Swap	2-Opt	Greedy	Nearest Insertion
Range = (0.10)	Improvement = 23.08%, time = 0.2s	Improvement = 85.01%, time = 7s	Improvement = 82.5%, time = 0.1s	Improvement = 69.79%, time = 1.38s
Range = (0.50)	Improvement = 22.46%, time = 0.23s	Improvement = 84.84%, time = 7.51s	Improvement = 82.28%, time = 0.11s	Improvement = 70.58%, time = 1.38s
Range = (0.200)	Improvement = 21.28%, time = 0.21s	Improvement = 84.38%, time = 7.46s	Improvement = 81.32%, time = 0.13s	Improvement = 70.75%, time = 1.36s

To conclude, out of these 4 heuristics: Swap, 2-Opt, Greedy and Nearest Insertion. The best tour reduction was almost always returned by 2-Opt, so if that is the most important thing then this algorithm will work best. However for larger numbers of nodes/cities 2-Opt may take a very long time and may not be worth the time, considering an algorithm like the greedy algorithm, returned results almost as good, but was always faster, especially for larger numbers of nodes. The swap heuristic was always limited in the tour reduction it was able to return but at least it always ran quite fast, but even despite this I would not say that the swap heuristic is particularly effective in minimising tour cost for the TSP problem. Finally my own algorithm which implemented the nearest insertion heuristic, was awful when there were very few nodes, giving even less optimal results than even the swap heuristic. Although it did improve significantly when there were more nodes, however it still did not result in a tour as optimal as the ones greedy or 2-Opt returned and it took longer to run then greedy. So overall I would say insertion heuristics are decent at reducing tour cost providing there are many nodes (at least 20) for it to be viable.