IADS Report

s1829305

March 2020

1 Algorithm

Overview

The main idea of the *Shortest Arc* heuristic algorithm is that we want the shortest paths between 2 nodes in our tour.

The first step in the algorithm is to compile a list of the distances between each of the nodes. We will store this as a tuple of length 3 with the distance between the 2 nodes and then each of the nodes as shown below:

$$\{(w(u,v),u,v)|u,v \in U, u \neq v\}$$

where U is the set of unvisited nodes.

We will loop through the list of unvisited nodes and take the minimum of this list, adding this arc (u,v) to another list and removing the corresponding nodes from the unvisited list.

We then are left with a list of $\lfloor \frac{n}{2} \rfloor$ arcs with possibly 1 node left over as unvisited. We then want to find the optimal order to put these arcs in the tour.

This is done in a similar way to the *Greedy/Nearest Neighbour* approach where we build up the tour by adding a starting arc, then loop through the remaining arcs, adding the closest one next. Note that we need to consider both (u,v) and (v,u) when choosing the next arc to add.

Finally we will append the leftover from the unvisited set, U, (if there is one) to the tour.

Running Time of the Shortest Arc Heuristic

Most of the loops for this algorithm run through the unvisited set but since $|U| \leq n$, for worst case runtime analysis we can use n.

Generating the list of arcs involves looping through the *self.dists* matrix and adding each distance with the corresponding nodes. This runs in $\Theta(n^2)$.

We then loop based on the length of the unvisited list, filter the arcs list to only contain unvisited nodes, find the shortest arc and then add this arc to the shortest arc list. Finding the minimum of a list l in python runs in $\Theta(|l|)$ and the length of the arcs list is $|U|^2$ so using our assumption from before this part of the algorithm runs in $O(n^3)$.

In the second part of the algorithm where we determine the order of the arcs in the tour we loop through the $\lfloor \frac{n}{2} \rfloor$ arcs adding the closest to the tour. For this we need to create a list of the distance of the arcs from the current end of the tour. This list is bounded by $O(n)$ as it is the length of the shortest arcs doubled (for the reverse of these arcs). We then also need to find the minimum of this which as before is $\Theta(n)$. This means that this part of the algorithm runs in $O(n^2)$.

Hence the full algorithm is $O(n^3)$.

2 Experiments

I ran some tests of the heuristic on randomly generated graphs. First I used graphs generated by sets of coordinates on a square of a given size. The straight line distances were used as the costs in this scenario. The results of one of these tests is shown below with other tests giving similar results.

```
Running tests using n=100, size=1000 with 20 graphs

Graph 0 : 83.4796293358791% cost reduction
Graph 1 : 80.27611467299269% cost reduction
Graph 2 : 80.03399191178764% cost reduction
Graph 3 : 80.82337257109764% cost reduction
Graph 4 : 80.41922810319458% cost reduction
Graph 5 : 79.43373827508483% cost reduction
Graph 6 : 81.89308254121642% cost reduction
Graph 7 : 79.28444021290038% cost reduction
Graph 8 : 81.44893216589813% cost reduction
Graph 9 : 78.77372761391912% cost reduction
Graph 10 : 76.93662785665757% cost reduction
Graph 11 : 76.46401940223419% cost reduction
Graph 12 : 77.06726417690679% cost reduction
Graph 13 : 80.01027939651381% cost reduction
Graph 14 : 78.16045832196798% cost reduction
Graph 15 : 81.22477998540641% cost reduction
Graph 16 : 76.83003784674376% cost reduction
Graph 17 : 77.98748534877417% cost reduction
Graph 18 : 80.71889432416285% cost reduction
Graph 19 : 80.69966867041659% cost reduction

Average reduction of 79.59828863668774%
[Finished in 5.9s]
```

Figure 1: Tests on Randomly Generated Euclidean Graphs

As the number of nodes decreases the algorithm seems to perform worse. I ran the test with various values of n and plotted my results as shown below:

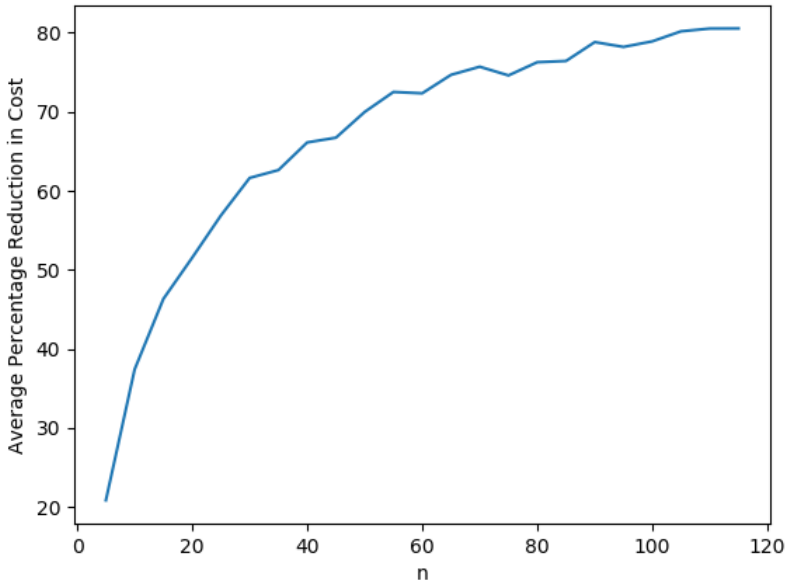

Figure 2: A Graph Showing the Reduction in Cost with various Numbers of Nodes

It is obvious from this graph that the algorithm works best on larger values of n with some levelling out in the performance gains around $n = 60$.

I then ran tests on randomly generated connected graphs using the same parameters as for the euclidean graphs.. The results are shown below:

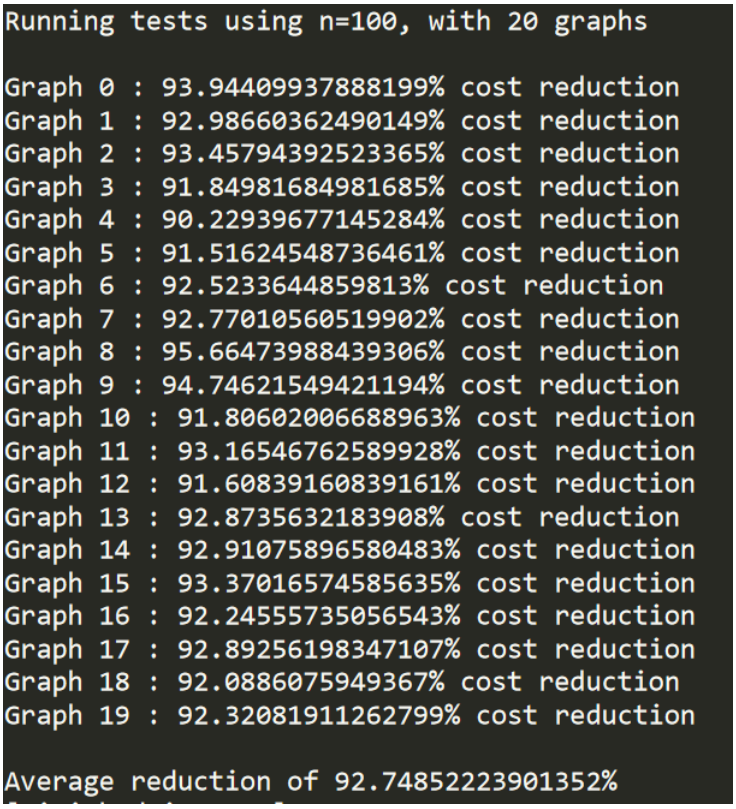

Figure 3: Tests on Randomly Generated Connected Graphs

The reduction here is higher than that of the euclidean graphs however this may simply be due to the way that they are generated as due to the edge costs being integers, many will have the same cost.

I then ran the same test as before, plotting the average cost reduction against the number of nodes in the graph. This gave me a graph with the same basic shape but slightly higher which reflects the better results as seen above. This graph is plotted below:

Interestingly the value that the graph starts to level out is lower at around $n = 30$ however again this is at a higher value than seen on euclidean graphs.

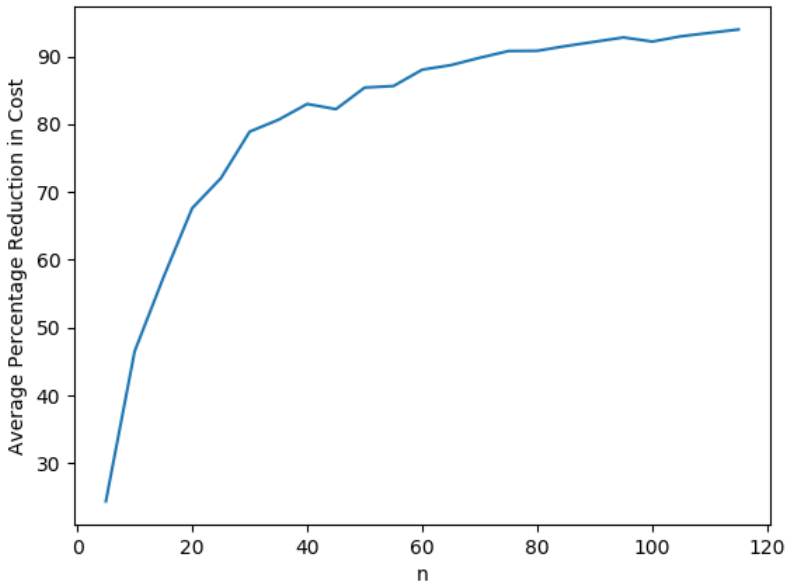

Figure 4: A Graph Showing the Reduction in Cost with various Numbers of Nodes

I also added a runtime option to the test functions so I could generate similar data to previous. These are shown below:

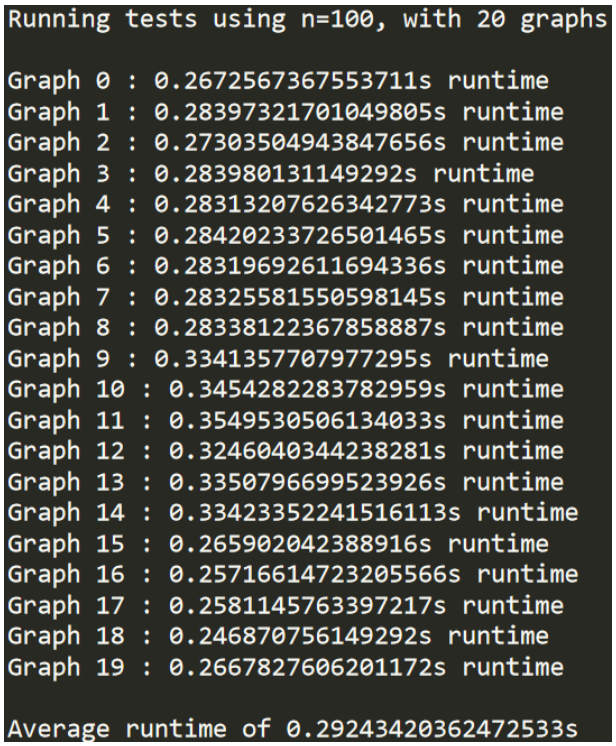

Figure 5: Runtime Readouts of the Algorithm

The runtimes here are quite small but as can be seen from the graph they increase quite rapidly. This is inline with the asymptotic analysis that I performed in the last section.

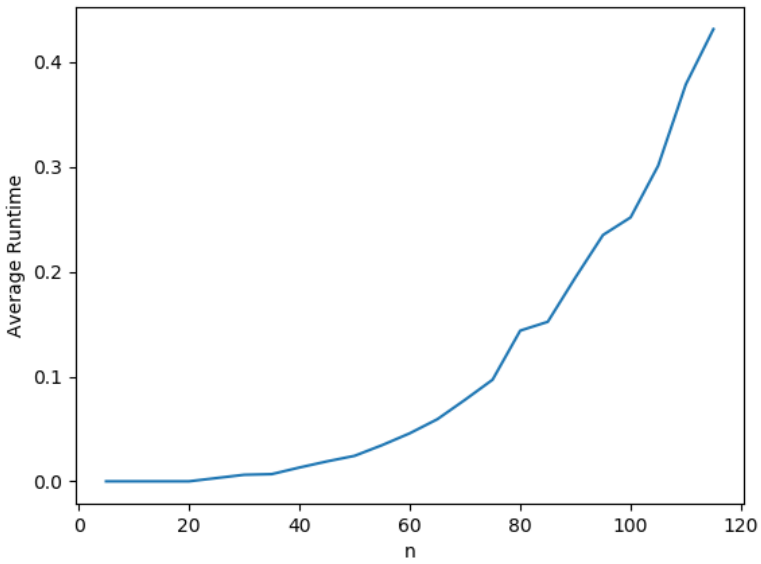

Figure 6: A Graph of Average Runtime for various Numbers of Nodes

From these tests I would conclude that the *Shortest Arc* heuristic is quite effective as for a large enough number of nodes it generally results in a tour cost reduction of $> 75\%$. For smaller graphs with fewer nodes this algorithm may be not worth it as it does not offer the same benefit in cost reduction with quite a high time complexity.