IADS Report – Travelling Salesman Problem

I. Algorithm

For part C of this coursework, I have implemented a tour improvement algorithm, called the 3-opt algorithm. A tour improvement algorithm starts with a complete tour and gradually tries to improve it to obtain an approximate solution closer to the best answer. This algorithm works similarly with 2-opt, but instead of disconnecting two edges, three are reconnected.

The implementation of my 3-opt algorithm is included in my graph.py file, made up of two methods: **ThreeOptHeuristic** and **tryReversePartC**. The high-level method ThreeOptHeuristic is similar to TwoOptHeuristic given for part B, where it generates all possible segment combinations. The effect of reversing these segments between will be considered by tryReversePartC, where reversal will be committed if it improves the tour value.

```
THREE-OPT-HEURISTIC():
1       Better ← True
2       while Better:
3           Better ← False
4           for i ← 0 to noOfCities - 1:
5               for j ← i + 2 to noOfCities - 1:
6                   for k ← j + 2 to noOfCities:
7                       if TRY-REVERSE(i, j, k):
8                           then Better ← True
```

```
TRY-REVERSE(i, j, k):
1       Flag ← False
2       assign b, e, f to Perm[i], Perm[j], Perm[k % noOfCities]
3       assign a, c, d to Perm[i-1], Perm[j-1], Perm[k-1]
4       OriginalDist = Dists[a][b] + Dists[c][d] + Dists[e][f]
5       calculate distances when edges are exchanged
6       check for each distances computed in 5
7       if it is less than OriginalDist, reverse the segment and set Flag to True
8       return Flag
```

In the pseudocode, *noOfCities* = self.n, *Perm* = self.perm and *Dists* = self.dists.
Here, the loop counter *j* and *k* start with index (i + 2) and (j + 2) respectively. This is because in TRY-REVERSE, we are going to get the (j – 1) and (k – 1) elements. For step 5, we are going to resemble the effect of exchanging edges by switching indexes to calculate the distance if a particular edge is exchanged. This will be then compared with *OriginalDist* to check if exchanging the particular edge improves the tour value. If it does improve, we

commit to reversing that particular segment. This is repeated for a different set of 3 edges until all possible combinations have been tried in a tour.

The time complexity of THREE-OPT-HEURISTIC is $O(n^3)$ because you need to check all combinations of 3 edges. Note that TRY-REVERSE takes O(1) time as it only contains assignments, arithmetic operations and comparisons. This heuristic gives better solutions than 2-opt (shown in Experiments section) but runs significantly slower since it checks all combinations of 2 edges instead of 3.

II. Experiments

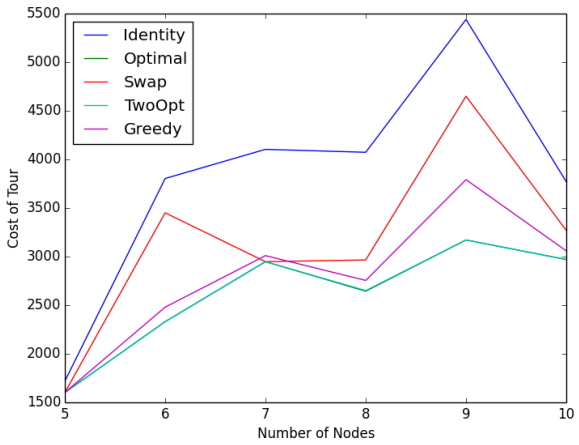
The results for given test files Cities25, Cities50, Cities75 and Sixnodes are given below.

Test File	Identity	Swap	2-opt	Swap & 2-opt	Greedy	MyAlgorithm
Cities25	6489.03543651	5027.40600646	2211.06628780	2233.01444669	2587.93344365	1993.08504258
Cities50	11842.55799220	8707.05639293	2781.27258226	2686.81369377	3011.59319197	2714.08489565
Cities75	18075.50674240	13126.07218710	3354.91215539	3291.07827326	3412.40531237	3272.02623611
Sixnodes	9	9	9	9	8	8

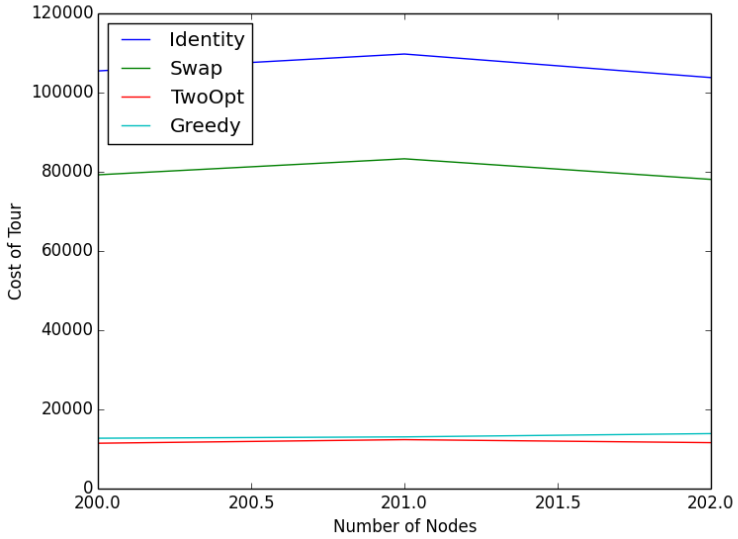
The 3-opt algorithm (MyAlgorithm) outperformed almost all cases of heuristics, except for Cities50, where doing a swap followed by 2-opt seemed to give slightly better solution. The particular part to look at is that 3-opt gives a better solution for all given test files than 2-opt.

I have implemented a method based on direct enumeration in graphs.py, called **optimal().** Its time complexity is O(n!), hence would only work for small number of nodes (<= 11).

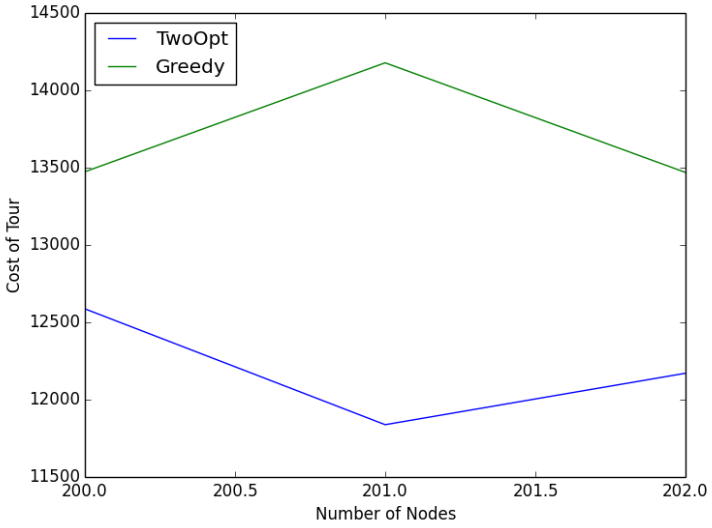
I have tried to test my algorithm with testMyAlgorithm method in tests.py but it would seem like it is 'stuck in an infinite loop', however when I run it in manually in graphs.py, it would work even for very large number of nodes.



For small number of nodes, it can be observed that 2-opt always returns the optimal solution. The next heuristic that is closest to the optimal solution is greedy, followed by swap. This is not always the case, i.e. number of nodes is 7, swap gives a solution better greedy.



For larger number of nodes, swap heuristic gives a solution that is nowhere close to greedy or 2-opt.



On a different random graph, this shows a clearer representation of 2-opt and greedy heuristic. On manually testing in graphs.py, 3-opt returns:

- For 200 nodes: 12015.1417874
- For 201 nodes: 11750.3705651
- For 202 nodes: 11548.8279056

This shows that 3-opt gives a better solution than 2-opt.