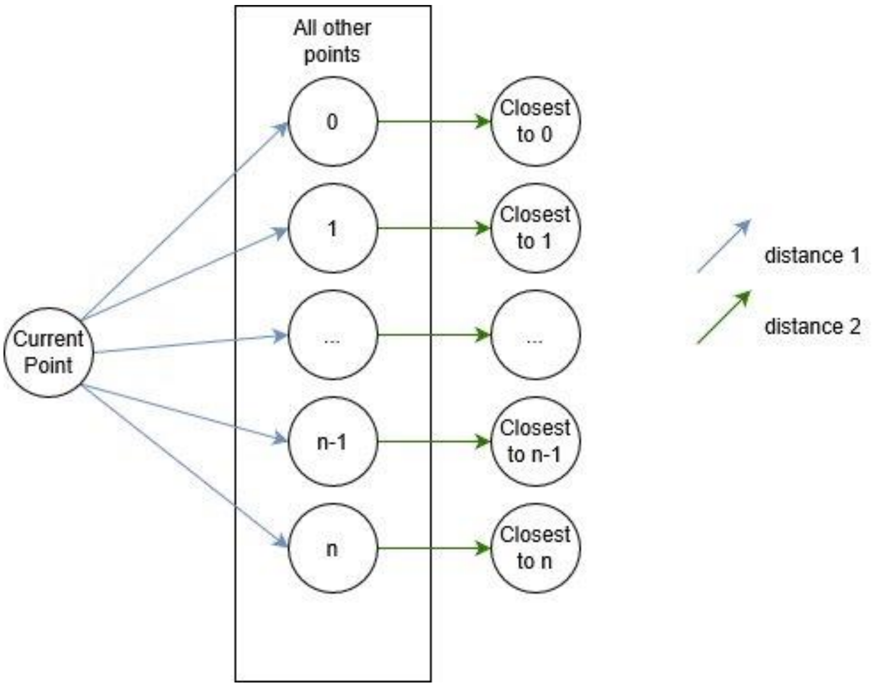
Informatics 2 – Introduction to Algorithms and Data Structures

Coursework 3: Heuristics for the Travelling Salesman Problem

Part C – Your Own Algorithm

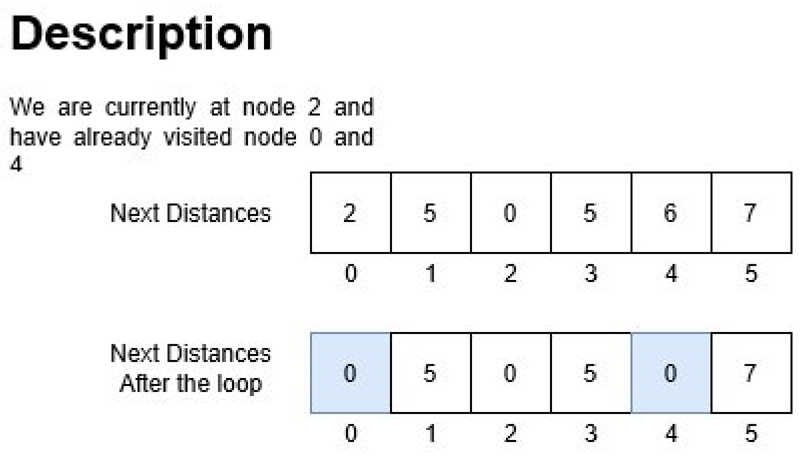
For my algorithm, I decided to implement a version close to Greedy. So, as we already know, Greedy looks at the point we're at, then looks at all the other non-visited points and chooses the closest one.

My algorithm, `LookAhead`, takes this a little further by looking two points ahead instead of just 1. It does that by looking at the point we're at and then looking at all the other reachable points. And then from each of these other points it takes the closest one. To get a better understanding of what it does here is a sketch:



Following the notation used on this sketch, what the algorithm then does is computing the sum of distance 1 + distance 2 and returns the minimum value. From there, it chooses the first point to go to. It's different than greedy because suppose we're at point 0 and the next closest point is 1 with distance 2, the current cost is 2. Now we go to 2 and the closest point is now 6 with distance 5, our cost is 2 + 5 = 7.But if the cost between 0 and 3 and between 3 and 5 was 3 and 2 respectively, we would have a total cost of 3 + 2 = 5. While Greedy would have chosen the first path, `LookAhead` would have chosen the second path.

Another challenge was to now make sure that we wouldn't go to already visited nodes or not move at all (because technically there is an edge from I to I with value 0). To do that, we have a special array called visited that keeps track of all the nodes we already have visited. So, when we get our list of next distances, we have a for loop that assigns the value 0 in our list of distances when a node is visited. (See sketch below)


Description
We are currently at node 2 and have already visited node 0 and 4
Next Distances
2 5 0 5 6 7
0 1 2 3 4 5
Next Distances After the loop
0 5 0 5 0 7
0 1 2 3 4 5

Now that this is done, we sum both the first and the second distance making sure that if distance 1 is 0, we do not sum it and leave the value as 0 (we do not need to have this constraint for distance 2 because when we first select the closest node, we make sure that it' the smallest non-zero distance). After this step, we have an array of sums which index correspond to the node we should/or shouldn't follow.

Another thing that comes to our attention in the example with the list before is that the smallest value can be the same. So, the rule that I introduced is just to pick the first occurrence of the number, it is arbitrarily chosen and could have been the last occurrence.

According to our list of sums we pick the smallest non-zero sum and get its index. The index corresponds to the next node to visit so we append it to the list of nodes to visit and we also append the node we were at in the list of already visited nodes.

When the length of the list of nodes we must visit reaches the number of nodes in our graph, we stop our algorithm and assign that value to self. perm.

I've then decided to call 2Opt on my final permutation. I've read some papers on a metaheuristic called GR-2Opt that hybridizes the greedy algorithm and the 2-opt method to solve the traveling salesman problem (TSP) and that is shown to be quite efficient. (source: https://ieeexplore.ieee.org/document/7872949). I have therefore made the choice to hybridize my LookAhead Algorithm as-well by adding a call to two-Opt.

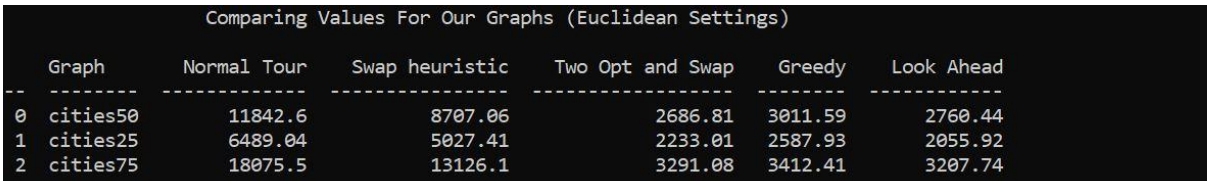
Part D – Experiments

Please before running tests.py, you need to install tabulate by typing these 3 commands:

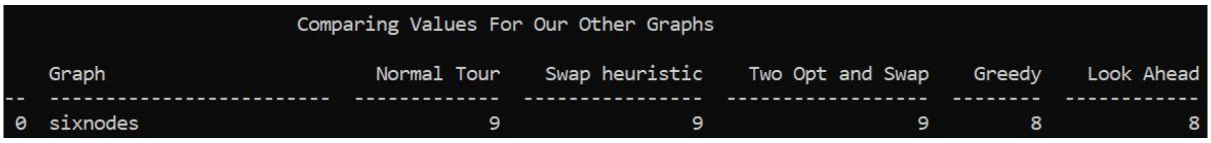
1° python3.6 -m venv env **2°** source ./env/bin/activate **3°** python3.6 -m pip install tabulate

For this part I first started by running our code for the graphs that we have been provided, that is for both the Euclidean case and the metric case. I had them in a table that shows what the Graph is, the normal Tour value, the tour value after a swap, the tour value after a swap + 2-opt, the tour value after Greedy and finally the tour value after using my own algorithm.

If you take a look at the code, all it does is create graphs of the form g = graph.Graph(-1,"name of file") for the Euclidian case and g = graph.Graph(NoOfNodes,"name of file") for the metric case. We then call the different heuristics and save the values to display. This gives the following results:



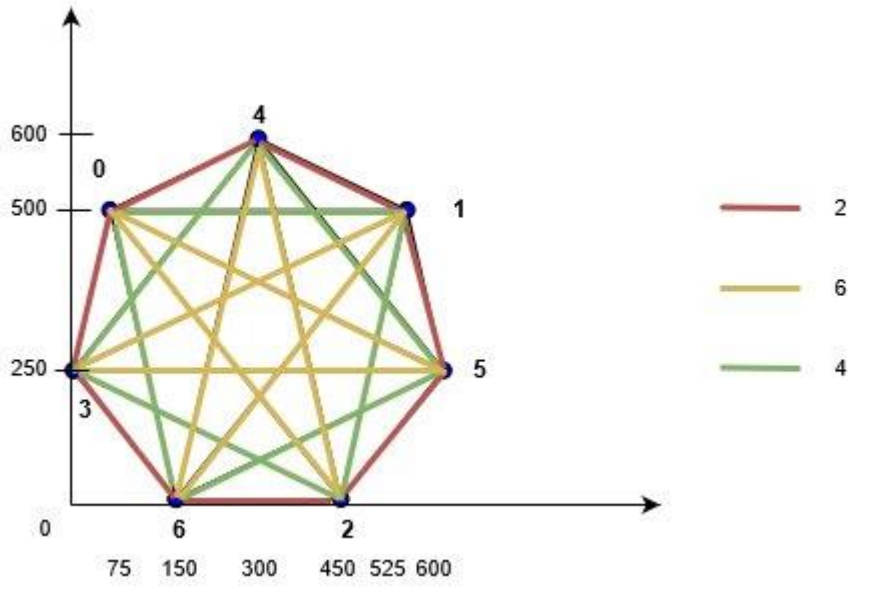
```
                 Comparing Values For Our Graphs (Euclidean Settings)

    Graph       Normal Tour    Swap heuristic    Two Opt and Swap    Greedy    Look Ahead
--  --------  -------------  ----------------  ------------------  --------  ------------
 0  cities50        11842.6           8707.06             2686.81   3011.59       2760.44
 1  cities25         6489.04          5027.41             2233.01   2587.93       2055.92
 2  cities75        18075.5          13126.1              3291.08   3412.41       3207.74
```



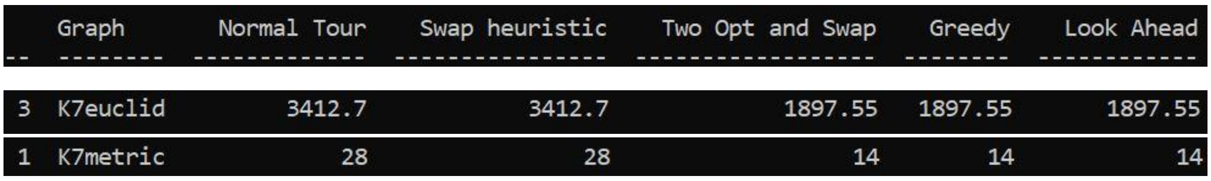
```
                         Comparing Values For Our Other Graphs

    Graph                       Normal Tour    Swap heuristic    Two Opt and Swap    Greedy    Look Ahead
--  ------------------------  -------------  ----------------  ------------------  --------  ------------
 0  sixnodes                              9                 9                   9         8             8
```

For this first set of tests, we can see that that every heuristic performs better than getting a normal tour value however if we get a closer look, only swapping isn't really close to an optimal solution since every other heuristic finds a way shorter path. From these observations only, we could suppose that overall, LookAhead performs better than Two-opt + Swap which itself performs better than greedy. However, for the case where we only have 50 cities, Two-opt + swap is the most performant.

To challenge the efficiency of Swap, I thought about the original ordering of the nodes having an important impact since swap only acts on adjacent indexes. For that I used this graph:

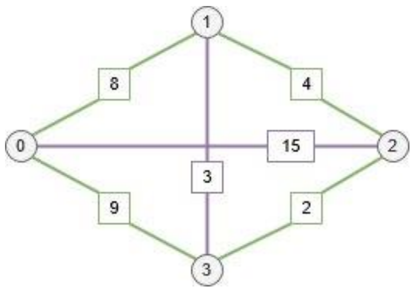


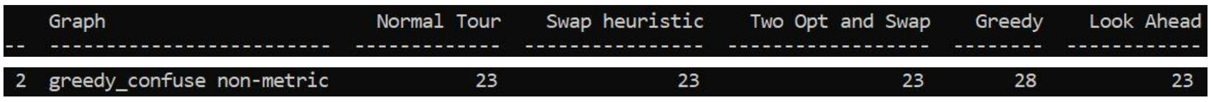
Because of the way we ordered the nodes, no swap is effective since the cost(for metric)/distance(for Euclid) would be the same, hence the heuristic won't give a different value than the normal tour, whereas all the other heuristics give the optimal value.



```
    Graph       Normal Tour    Swap heuristic    Two Opt and Swap    Greedy    Look Ahead
--  --------  -------------  ----------------  ------------------  --------  ------------

 3  K7euclid         3412.7            3412.7             1897.55   1897.55       1897.55

 1  K7metric             28                28                  14        14            14
```

Then I thought about challenging Greedy in the non-metric case. I designed a Graph where greedy will try to minimize the cost of the first step but ends up taking a longer path in the end. Here is what the graph looks like and the result of running all heuristics on that non-metric graph.

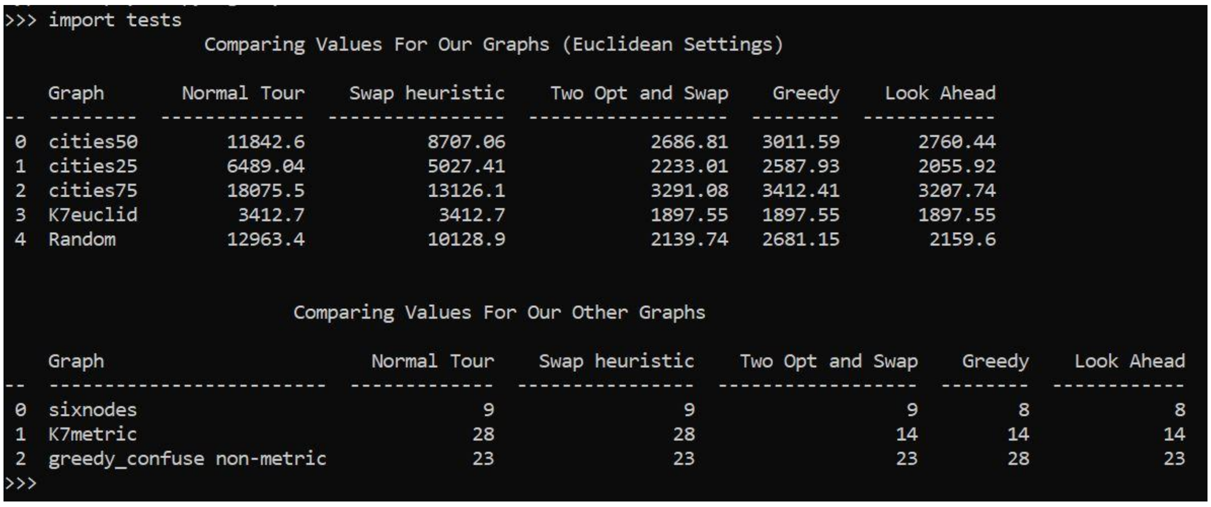




```
   Graph                      Normal Tour    Swap heuristic    Two Opt and Swap    Greedy    Look Ahead
-- ------------------------  -------------  ----------------  ------------------  --------  ------------
 2  greedy_confuse non-metric            23                23                  23        28            23
```

As we can see, while all the other heuristics give the optimal solution, Greedy picks the longest path by making a bad first choice. My algorithm Look Ahead, is designed such that these problems are avoided most of the time. Hence in some cases Greedy fails to find an optimal solution because it doesn't check a lot of the available information we have on the graph. In the graph that I constructed, it actually takes the worst route, and it's easy to see that for non-metric cases we could come up with a few more graphs (more complex even) like these where greedy will pick the worst path. If we take a look at the provided graphs (cities25, cities50 and cities75) we can come up with a similar conclusion. If we compare the value of Two-opt + swap with Greedy, we get further away from a more optimal solution as we increase the number of cities. To improve this side of the algorithm, we could call two-opt on it and eventually improve the final tour. LookAhead is also in a way a Nearest neighbour algorithm however it takes into consideration more data because it looks at 2 points instead of 1 but also because it ends with a call to 2-opt that then takes even more possibilities into consideration.

Finally, my last test was generating 100 random x y coordinates (with respect to the triangle inequality). *(NB: this will create a file)* The final output of tests.py is:



```
>>> import tests
                 Comparing Values For Our Graphs (Euclidean Settings)

   Graph       Normal Tour    Swap heuristic    Two Opt and Swap    Greedy    Look Ahead
-- --------  -------------  ----------------  ------------------  --------  ------------
 0  cities50        11842.6           8707.06             2686.81   3011.59       2760.44
 1  cities25        6489.04           5027.41             2233.01   2587.93       2055.92
 2  cities75        18075.5           13126.1             3291.08   3412.41       3207.74
 3  K7euclid         3412.7            3412.7             1897.55   1897.55       1897.55
 4  Random          12963.4           10128.9             2139.74   2681.15        2159.6


                         Comparing Values For Our Other Graphs

   Graph                      Normal Tour    Swap heuristic    Two Opt and Swap    Greedy    Look Ahead
-- ------------------------  -------------  ----------------  ------------------  --------  ------------
 0  sixnodes                              9                 9                   9         8             8
 1  K7metric                             28                28                  14        14            14
 2  greedy_confuse non-metric            23                23                  23        28            23
>>>
```

As we can see, Two-opt gives the best answer out of all the heuristics. So we can assume from the tests we did, that 2-opt will generally perform better than the other heuristics we have and that Nearest neighbour approaches (while time efficient) are improved when adding a call to two opt which explains the efficiency of LookAhead.