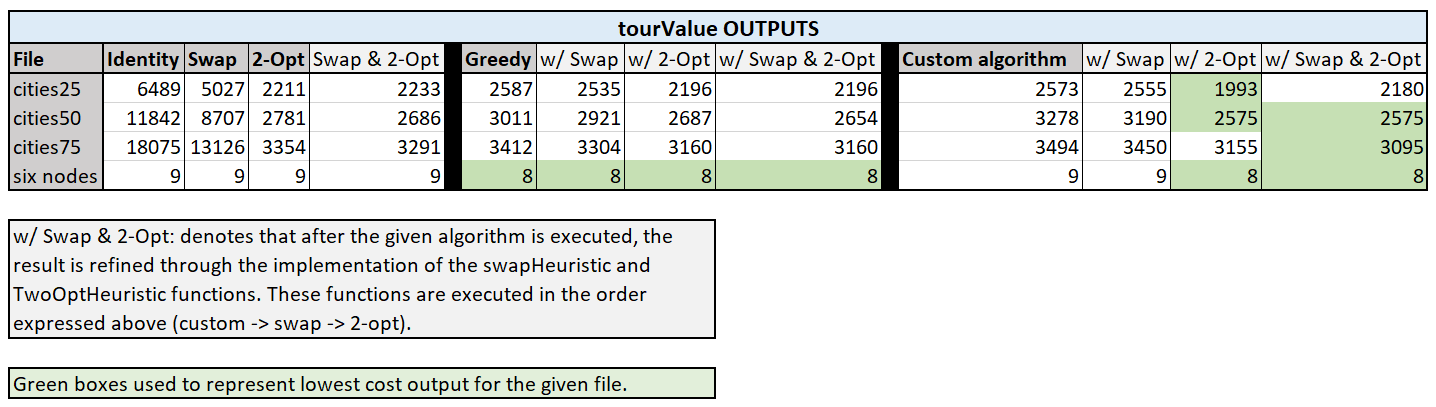
**IADS Coursework 3 Report  
Heuristics for the Travelling Salesman Problem**

**My custom algorithm ‘Temperate’**  
I formulated my own custom algorithm for finding approximate optimal routes in polynomial time.

**How it works**  
I wanted to create an algorithm that finds optimal paths by prioritizing the cities/nodes which have the highest mean distances from other cities. This prioritization works by creating fragments/edges (represents the transition between two cities/nodes) between the city/node with the highest mean distance and expanding it with its best possible path (shortest). This is iterated for all cities from highest to lowest mean distance and placed appropriately in a priority queue (descending order of mean values) until all nodes are used up. Once a priority queue of fragments/edges is found then the route can begin to be created. The route is initialized with the first fragment in the priority queue. Thereafter node expansion is based upon availability in the priority queue or best possible transition from the given node. By this I mean that based upon the given the node we need to expand (last node in current route), the algorithm first checks if this node can be found in a fragment in the priority queue. If so, the nodes for this given fragment are added to the route. Otherwise, the algorithm finds the best available transition for the given city/node and adds it to the route. This process is continued until a complete route is formed.

In order to ensure consistency throughout this process lists are used to represent all nodes/fragments that have already been explored.

**Overall thoughts**  
I realized upon implementation it almost works in the opposite way to the greedy algorithm in which rather than prioritizing using the shortest distances possible, mine prioritizes **not** using the longest distances possible, this is why I decided to name this algorithm ‘Temperate’(the antonym of greedy). Overall, I thought this concept would be effective as upon this prioritization, the most significantly large distances/costs can be reduced to their minimum possible value.

**Results**  
Generally my results were on the same par as the Greedy algorithm (on average a 3,6% higher tourValue), which I was happy with given the similarities and disparities between their associated prioritization techniques. However, what I found particularly exciting was the results of my algorithm after being parsed by the refinement algorithms Swap and 2-Opt. The results upon parsing the result of my algorithm with the combination of these refinement algorithms produced the lowest cost routes throughout all given implementations.

**Experiments**  
I formulated my own functions that generate random Euclidean/Metric graphs in order to test and compare the efficiency of different algorithms at a large scale. (All code within tests.py)

**Generating random graphs:** *createRandomMetricGraph(),* *createRandomEuclideanGraph()*  
In order to generate random graphs that could be used as input to any of my algorithms I decided to generate them individually in the form of a text file. The given graph size to generate is specified by the size input, and the cost of edges for a given graph was calculated by choosing a random number between the upper and lower bound inputs.

**Comparing efficiency of different algorithms using random graphs:** *calculateCostDiffs()*  
In order to compare the efficiency of different algorithms we can compare how well they do in calculating the tourValue for a given graph. To make this comparison accurate and reliable we must do this at a large scale by comparing these algorithms multiple times with different graphs. So, to achieve this, I created a function which compares the efficiency of two algorithms by calculating the mean difference in their tourValue for **n** tests (n = the input value for the desired number of tests). This function accounts for both Euclidean and Metric graphs.

In order to ensure high accuracy and reliability all outputs below were produced using the average of 500 random tests for Euclidean/Metric graphs of the specified size.

A screenshot of a computer

Description automatically generated