Algorithms: Travelling Salesman Problem

Pseudocode:

```
read in command arguments for input coordinates, output tour, and max time
store x and y values from input coordinates into an n by 2 matrix
calculate Euclidean distance between each coordinate, giving an n by n matrix of dist
while time passed is less then max time:
        # Random Nearest Neighbor
        start a new tour at a random node
        while there exists unvisited nodes:
                from last node in tour, find the 2 closest non visited nodes
                at random, pick one of the 2 nodes as the next node
                put this node as the last node in the tour
        # 3-Opt
        while nodes still switched for entire loop through tour:
                pick 3 nodes
                find the order of the 3 nodes that results in the shortest path
                change the tour to have that path instead
        calculate cost of tour
        if tour has a shorter cost than the best tour:
                best tour becomes current tour
                best cost becomes current cost
write to output tour file best cost and best tour
return best tour's first node
```

Rationale:

Since the assignment asks to return the best tour within a certain time, I thought the best way to implement the problem would be to keep a best tour over as many iterations as possible. At first, I thought about starting with a random tour, then implementing 2-opt to loop through the tour and swap 2 nodes if they result in a lower cost. This did not well at all because as I learned, 2-opt looks for a local solution, so it is based on the initial tour before optimizing. So instead, I thought of how I could make the starting tour better and realized I could start with a greedy solution and optimize that. At first, I used Nearest Neighbor to start with a better tour before 2-Opt, but this limited me to only n number of starting tours, 1 for each starting node. I then decided instead of picking the closest node, to increase variance in starting nodes, I could find the 3 closest nodes and at random pick one of them. This brought me even closer to the optimal solution. Next, to test whether more variance was resulting in better tours, I brought down the number of nodes to choose the 2 closest nodes, and this in fact brought down the average cost. Lastly, instead of implemented 2-Opt, I decided to add 1 more node for swapping, so instead of swapping 2 nodes and seeing if it results in a better tour, swapping 3 nodes and finding the best configuration out of the 3. After this last optimization, my algorithm now finds the optimal solution of the example nodes, mat-test.txt, just about every run.