CS 325 Section 401, Project Group 1
David Rigert, Isaiah Perrotte-Fentress, Adam McDaniel

Project 4: Traveling Salesman Problem (TSP)

Algorithm Description

Algorithm Name: Repetitive Nearest Neighbor

We tried multiple algorithms during testing including: minimum spanning tree algorithm, ant colony algorithm, and brute force (for entertainment value). We determined that brute force and ant colony took considerably too long as the number of nodes grew. We found that the minimum spanning tree ran quickly, but failed to find a solution within 1.25 of the optimal for our test cases.

After those missteps, we settled on Repetitive Nearest Neighbor as the most effective algorithm for our test cases. The Nearest Neighbor algorithm is effective at finding an answer quickly, and due to its efficiency, can be run multiple times to repetitively seek the minimum tour length.

Before the algorithm runs, some pre-work is done. Our program reads from the input file and stores the coordinates and node ID in a vector. Our program also initializes a 2D array to house all of the edge weights between all possible combinations set to 0. This was done to potentially speed up the program depending on the algorithm we implement. Eventually, all of the edge weights will be filled in as a node searches for the nearest neighbor.

Once the algorithm has its starting information, a starting node is randomly selected. The algorithm searches for the nearest neighbor to that node, and creates a connection (storing the path followed for later reference). This process continues, being sure not to retrace through any nodes that have already been visited. This continues until only one node remains, and the circuit is complete. Afterwards, the results of the path followed and the edge weights are stored in variables. If this is the first run the results are kept. For every run after, the results are compared to the previous best, and only the minimum of the two are kept. This process repeats through all cities until all have been used as the starting node, or 270 seconds of runtime has been hit (max allowed for the assignment with a 30 second buffer).

Since this program performs similar tasks multiple times with the only change in the starting node, we decided to make use of multi-threading. By using multi-threading, we were able to greatly reduce the runtime by having multiple instances of the algorithm run at once. This allowed us to to check additional nodes as starting points on larger problems, resulting in a slightly more efficient circuit.

Once the results have been retrieved the path taken and resulting tour length are exported to a file named <input file name>.tour.

We looked at a number of algorithms outlined here: https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf , and referenced the following document as well: http://www-eio.upc.es/~nasini/Blog/TSP_Notes.pdf

Pseudo Code

```
Inputs:
        cities
Variables:
        bestTour
        bestTourPath
        visitedNodes
        minLength
        tourLength
        maxTime = 270 (seconds)                          // Max allowed for this assignment is 5 min,…
        currentNode = 0th node (first node)              // …includes a 30 second buffer

visitedNode adds currentNode

While time > 0                                           // Loop till 3 minutes is hit
        While visitedNode.count does not equal cities.count   // Loop till circuit created
                For X nodes not in VisitedNode           // Loop through all nodes not yet visited
                        CalculatedEdge = X distance to currentNode
                        If X < minLength                 // Check to see if nearest neighbor
                                minLength = X

                visitedNode adds node X                  // Add nearest neighbor to visited
                tourLength = tourLength + minLength      // Add weight to tour
                currentNode = node X                     // Update the new current node

        // This completes the circuit after the while loop
        tourLength = tourLength + currentNode distance to originalNode

        // Check to see if this was the best tour
        if tourLength is less than bestTour
                // The order of visited nodes provides the path followed
                bestTour = tourLength
                BestTourPath = visitedNodes

        // Calculate time remaining to try more vertexes
        Calculate timeElapsed
        maxTime = maxTime - timeElapsed

// Prints out results as specified in the assignment
Print to file bestTour and bestTourPath
```

Best Tours for Example Instances

The table below contains our best runtimes and best tours for the Example TSP Cases provided. We have included txt.tour files of our best runtimes.

File Name	Tour Time	Tour Length
tsp_example_1	0 seconds	130921
tsp_example_2	1 second	3118
tsp_example_3	4:38	1929250

Best Tours for Competition Test Instances

The table below contains our best tours for the Competition TSP Test Cases provided. We have included txt.tour files of our best runtimes.

File Name	Tour Length
test-input-1	5778
test-input-2	8011
test-input-3	14851
test-input-4	19560
test-input-5	27292
test-input-6	38905
test-input-7	61577