Project Group 12

Billy Ji (Student ID 932764814)
Andrew Calhoun (Student ID 932709292)
Matthew Hillman (Student ID 932505959)

CS 325 - 400 (Spring 2016)
Instructor: Julianne Schutfort

REPORT - PROJECT 4
Traveling Salesman Project

1.Three algorithms for solving the Traveling Salesman Problem are Christofides', 2-Opt, and nearest neighbor.

Christofides' algorithm takes advantage of the fact that for a given Eulerian graph, you can find a Eulerian path in O(n) time. So given a TSP problem, you first find a minimum spanning tree T for this problem. You then create an Eulerian graph by finding the minimum cost matching on odd degree nodes. If you add the matching edges, this makes the degree of all nodes even and thus creates an Eulerian graph. Now that all edges have even degree, we can find an Eulerian tour. During this tour, whenever a vertex that has already been reached is encountered, we skip this edge, and instead connect the current edge to the next new vertex. The end tour will have the distance. However, the drawback to this algorithm is that it is often not accurate, giving a result that is at most 1.5 times the optimal distance.

Another algorithm we looked at is the 2-Opt algorithm. The basic idea behind this algorithm is that when it finds a route in the TSP that crosses over itself, it will reorder the nodes so that it doesn't. So the 2-Opt algorithm will consider transposition of two nodes at a time. Let's say it starts with nodes 1 and 2. If transposition of these two nodes results in a smaller value than that of the initial solution, the value is stored away for future consideration. Otherwise the value is ignored and the next two nodes are considered(e.g. nodes 1 and 3). This is carried out until all pairwise exchanges have been considered. Given a TSP with n nodes, each node can be considered with n-1 other nodes and since there are n nodes in total, there are n(n-1)/2 different exchanges. 2-Opt is fairly quick(for solving the TSP problem) and returns close to optimal routes, especially when paired with other high-efficiency algorithms that have already calculated the path.

A third algorithm is the nearest neighbor algorithm, which is a greedy algorithm that runs fairly quickly but returns an solution route that is often not optimal, and can even return a route that traverses the max possible distance. First, a random vertex is picked. From this vertex, we travel to the nearest unvisited vertex. We mark this vertex as visited, and from there find the nearest unvisited vertex. This process is repeated until all vertices have been visited.

2.
Our take on this problem was to first run the nearest neighbor algorithm, and then the 2-Opt algorithm twice to optimize and make up for the nearest neighbor algorithm's inaccuracy.

The nearestNeighbor file has three functions, one to initialize the adjacency matrix, one to run the nearestNeighbor tour, and one to get the total cost of the tour.

The first function, randomAdjacencyMatrix, takes in the number of cities, and creates an adjacency matrix by initializing a vector of vector<int>s that has the size of the number of cities. Each value in the adjacency matrix is set to a random number using the rand() function.

The second function, nnTour, takes in the initialized adjacency matrix, as well as the number of cities. Variables are initialized for currentCity(initialized to 0), nextCity, and minimumDistance.
A vector<int> called tour is initialized with size (number of cities +1) and all values as -1.
A nested for loop runs, with both loops(using variables i and j, respectively) going from 0 to the number of cities.

In the first for loop, minimumDistance is set to an arbitrarily large number and tour[i] is set to currentCity.
In the second for loop, if j does not equal currentCity and adjacencyMatrix[currentCity][j] is less than minDistance and j is currently at the last position in the tour vector, we update minimumDistance to the value at adjacencyMatrix[currentCity][j] and update nextCity to j. currentCity is then updated to nextCity, and the for loop runs again.

At the end of this function, the last value of tour is set to 0 and the vector is returned.

The third function, getCost, takes in the adjacency matrix as well as a vector tour that should hold the order that cities are to be visited in. Variables are initialized for cost, totalCost(initialized to 0), tourLength(initialized to tour.size()), currentCity, and nextCity.

A simple for loop is run from 0 to tourLength-1, to add up all the values in the adjacency matrix according to the order that is indicated from vector tour. At the end totalCost is returned, and this is the distance that was found.

The 2opt file has two functions: the main function, twoOpt, to run the 2-Opt algorithm and a second helper function, swapOpt.

The main function takes in first initializes a bool variable improve to false, and a vector solutionTour to nnTour.

The helper function takes in a vector of ints(nnTour) that has had the nnTour function run on it, two ints swapFront and swapEnd, and the number of cities. We initialize a vector newTour to nnTour.

To gain additional accuracy, we ran the original 2-Opt Tour through yet another 2-Opt Tour to further reduce the distance if at all possible. While this did not happen in every instance, on shorter tours, the difference was often upwards of 100 or 200 units of distance.

3. A discussion on why you selected the algorithms

The nearest neighbor tour algorithm is a greedy algorithm and was one of the first and most reliable methods of finding a solution to the traveling salesman problem. It will visit the nearest city repeatedly until all of the cities are visited and while it can generate a fairly short (low-cost) tour, however, it is rarely the optimal tour. However, given that greedy solutions can sometimes create optimal solutions as well as give good starter data sets they can provide a good foundation for results when combined with another, more precise, but potentially costly algorithm.

The second algorithm, which we ran twice, is the 2-Opt. It compares every possible valid combination and evaluates them under a swapping mechanism. It is used to remove any route cross-over to increase the efficiency of the solution and reduce the total distance. We ultimately chose this algorithm because it was more time effective than other alternatives, such as Christofides, which we also evaluated, but felt would not garner us the results we wanted in less than 3 minutes.

In some cases, the 2-Opt improvement was minimal if the nearest neighbor already found a near-optimal solution when

These combinations proved early to give approximation well within Rho <= 1.25 and a few tweaks helped get the first two examples below rho <= 1.10 in some instances.

4. Nearest Neighbor pseudo-code:

```
Vector of Integer Vectors randomAdjacencyMatrix(numberOfCities)
        create vector of integer vectors adjacencyMatrix
        resize adjacencyMatrix to numberOfCities

        i <- 0
        j <- 0
        For i to numberOfCities
                For j to i
                        randomNumber <- rand() % 200 + 1;
                        if i equals j
                                Set randomNumber to -1;
                        adjacencyMatrix[i][j] = randomNumber;
                        adjacencyMatrix[j][i] = randomNumber;


        return adjacencyMatrix;


Vector Integer nnTour(vector of integer vectors adjacencyMatrix, numberOfCities)

        Create integer vector tour
        Resize tour to numberOfCities+1, initializing all values to -1

        i <- 0
```

```
        j <- 0

        For i to numberOfCities

                minDistance <- infinity
                tour[i] <- currentCity
                For j to numberOfCities

                        If j does not equal currentCity and adjacencyMatrix[currentCity][j] is less than
                        minDistance and j is currently at the last position in tour
                                minDistance <- adjacencyMatrix[currentCity][j]
                                nextCity <- j

                currentCity <- nextCity

        tour[numberOfCities] <- 0;
        return tour;


getCost(vector of integer vectors adjacencyMatrix, integer vector tour)
        tourLength <- tour size
        Initialize totalCost to 0
        For 0 to tourLength-1
                currentCity <- tour[i]
                nextCity <- tour[i+1]
                cost <- adjacencyMatrix[currentCity][nextCity]
                totalCost <- totalCost + cost
        return totalCost
```

2-Opt pseudo-code

```
Integer Vector twoOpt(Integer Vector nnTour, Vector of Integer Vectors adjacencyMatrix,
numberOfCities, timer)

        size <- nnTour size
        improve <- false
        i <- 0

        integer vector solutionTour <- nnTour
        maxTime <- CLOCKS_PER_SEC * 15     // this is a diminishing return brake. Results do not
improve markedly or become much more efficient after 15 seconds. -- CPS is approx. 1,000,000.
        While improve is false and timer is less than maxTime
                bestDistance <- getCost(adjacencyMatrix, solutionTour)
                For i to size-2
                        For k=i+1 to size
                                Integer vector newTour <- swapOpt(solutionTour, i, k, numberOfCities)
                                newDistance <- getCost(adjacencyMatrix, newTour)
```

```
                                        // improvements found, go back
                                        If newDistance less than bestDistance
                                                improve <- false;
                                                solutionTour <- newTour
                                                bestDistance <- newDistance
                                        curTime <- clock()
                                        loopTime <- curTime - timer
                                        If loopTime greater than maxTime
                                                return solutionTour
                        improve <- true
                return solutionTour


Integer vector swapOpt(integer vector nnTour, swapFront, swapEnd, numCities)
        size <- size of nnTour
        Integer vector newTour <- nnTour

        For 0 to swapFront-1
                newTour[i] = nnTour[i]
        dec <- 0

        i <- swapFront

        For i to swapEnd
                newTour[i] <- nnTour[swapEnd - dec]
                Increment dec by 1

        i <- swapEnd + 1

        For i to size
                newTour[i] <- nnTour[i]

        return newTour


Traveling Salesman pseudo-code
getInput(fileName, vector of integer vectors adjacencyMatrix)

        Initialize Vector of Cities cityList   // City is a struct that contains data for coordinates
        Open fileName using inputFile
        While lines can be read from inputFile into readLine
                iss -> inputCity.id
                iss -> inputCity.x
                iss -> inputCity.y
                Add inputCity to cityList
        Close inputFile

        numberOfCities <- size of cityList
```

```
Resize adjacencyMatrix to numberOfCities, initializing all values to -1

i <- 0

For i to numberOfCities
        For j to numberOfCities
                // distance formula, rounded to nearest integer -- d = sqrt(((x2-x1) + (y2-y1))^2)
                adjacencyMatrix[i][j] <- round(sqrt(pow((cityList[i].x - cityList[j].x), 2) +
                pow((cityList[i].y - cityList[j].y), 2)));
```

5. Your best tours for the three example instances, and the time it took to obtain these tours

	NN-Distance	Double 2-OPT Refinment Distance	NN Rho	Double 2-Opt Refineme nt Rho	Time To Obtain Tours
Example -1 - .01 Second	150393	118293	1.39	1.09	0.01 seconds
Example 2 - 12.47 Seconds	3210	2822	1.24	1.09	12.47 seconds
Example 3 - 50.48 seconds	1964948	1964948	1.25	1.25	50.48 seconds

6. Your best tours for the competition test instances.

Test Case / Seconds	Nearest Neighbor	2x 2-Opt-After NN	Time In Seconds To Find Tour
Test Case 1	5926	5639	0.006 Seconds
Test Case 2	9503	7788	0.48 seconds
Test Case 3	15829	12988	7.78 seconds

Test Case 4	20215	19820	15.01 seconds
Test Case 5	28685	28581	15.02 seconds
Test Case 6	40933	40926	15.04 seconds
Test Case 7	63780	63780	15.21 seconds