David Kopp

9-26-15

Comp496 ALG

Project #2: Euclidean Traveling Salesman Problem

David Kopp

11-2-2015

Comp496ALG

Project #2: Euclidean Traveling Salesman Problem

Run Time Analysis of Brute Force Algorithm:

The run time for the brute force solution of the Euclidean traveling salesman problem grows very quickly for every n added. See the graph below for run time execution of n = 5, 7, 9, 12.

The brute force solution always gets the shortest tour, but it takes the longest to find.

Big O Analysis: For the brute force solution the permutation method is driving the execution times upwards.

By Methods:

ReadInputFile() : O(C + 4n + n^2)

-Setting size of data structures and initializing a variable: constant = O(C)

-Initializing Arrays: O(4n)

initArray(): O(n)

-Loop – O(n)

initPermutationArray(): O(n)

-Loop – O(n)

-Inputting data into distanceTable[][] – O(n^2)

bruteForceSolution(): O(n^2 + n + 2C)

calculateDistances(): O(n^2)

initArray(): O(n)

-Initialize variables – O(C)

checkDistance(): O(C)

permutations(): lexicographic Permutation Algorithm C \* O(n^2\*n!)

checkDistance(): – O(C)

Total = O(C + 5n + 2n^2 + C(n^2\*n!))

Final Analysis = O(n^2\*n!)

Run Time Analysis of Greedy Algorithm:

The run time for the greedy solution of Euclidean traveling salesman problem is that it is very linear and fast. See the graph below for run time execution of n = 5, 7, 9, 12, 26.

The greedy algorithm gets a longer path then that of the brute force, but it is many times faster for a larger n.

Big O Analysis: For the greedy solution most of the calls where O(n) in nature.

By Methods:

ReadInputFile() : O(C + 4n + n^2)

-Setting size of data structures and initializing a variable: constant = O(C)

-Initializing Arrays: O(4n)

initArray(): O(n)

-Loop – O(n)

initPermutationArray(): O(n)

-Loop – O(n)

-Inputting data into distanceTable[][] – O(n^2)

ETSPGreedy(): O(C + 3n + 2n^2)

Initializing objects – O(C)

initArray: O(n)

calculateDistances() – O(n^2)

greedyTour: O(n^2 + n)

computerDistance: O(n)

Total = O(2C + 7n + 3n^2)

Final Analysis = O(n^2 + n)

Run Time Analysis of Minimum Spanning Tree Algorithm:

The runtime analysis for the minimum spanning tree solution that uses a depth first search to find the shortest tour is a little slower in execution than the greedy algorithm, but its very negligible. See below for run time execution of n = 5, 7, 9, 12, 26.

The minimum spanning tree solution is much faster than brute force and also gets a close on the tour distances. While MST takes longer then greedy it gives a shorter tour distance.

Big O Analysis: For the mst solution we used prim’s solution and a dfs on the mst to get the shortest tour. The prim’s algorithm is the same one from class, so we already know the big O.

By Methods:

ReadInputFile() : O(C + 4n + n^2)

-Setting size of data structures and initializing a variable: constant = O(C)

-Initializing Arrays: O(4n)

initArray(): O(n)

-Loop – O(n)

initPermutationArray(): O(n)

-Loop – O(n)

-Inputting data into distanceTable[][] – O(n^2)

ESTPSMinSpanTree(): O(C + 2n + 2n^2 + m)

Initializing objects – O(C)

initArray: O(n)

calculateDistances() – O(n^2)

createGraph: O(n^2 + m)

computerDistance: O(n)

Total = O(2C + 6n + 3n^2 + m)

Final Analysis = O(n + n^2 + m)

Tour Scatter Plot of Instructor Test Case #2: (MST Solution)

MST Tour: 12 Cities

Min Cost 628.7697297775677

0 2 10 8 3 4 6 11 7 5 1 9

A C K I D E G L H F B J A

Run Time was 38msecs

Conclusion:

In conclusion, the brute force solution gave the shortest tours, but was O(n^2\*n!). In addition, the brute force solution is not as optimal as it sounds as the more n the longer it takes to run at a very steep cost in execution time. The greedy solution in comparison to brute force gave the longest tour routes, but was the fastest runtime at O(n^2 +n). Unlike brute force the greedy solution can be used for a much large n without sacrificing execution time. The best and most preferred solution is the minimum spanning tree, or mst, with the depth first search to find a tour that is very close as being as short as brute force. The mst solution was much better than the greedy solution as well in shortest distance. Also, the mst solution is very fast almost the same speed as the greedy solution at O(n + n^2 + m). Below you will can see a table that shows the greedy and mst that was done by hand and verified to be the correct path. Both the greedy and mst where accurate. It is worth mentioning that there is more than one tour for the mst and brute force solution that will give you the same distance of a tour. There are more in the brute force solution and only a select few in the mst. The more cities creates a larger number of same distance combinations.

Distance Table for instructor test case #1: n= 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Dist. Table | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 34.83 | 55.01 | 71.28 | 37.59 |
| 1 | 34.83 | 0 | 43.28 | 39.22 | 14.21 |
| 2 | 55.01 | 43.28 | 0 | 41.19 | 29.07 |
| 3 | 71.28 | 39.22 | 41.19 | 0 | 34.06 |
| 4 | 37.59 | 14.21 | 29.07 | 34.06 | 0 |

Greedy Solution (Starting at 0):

0, 1, 4, 2, 3, 0

Distance = 190.58

(This approximately matches the greedy solution)

MST Solution (Starting at 0):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MST | 0 | 1 | 2 | 3 | 4 |
| D | 0 | 34.83 | 55.01 | 71.28 | 37.59 |
| P | - | 0 | 0 | 0 | 0 |
| S | T | F | F | F | F |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| D | 0 | 34.83 | 43.28 | 39.22 | 14.21 |
| P | - | 0 | 1 | 1 | 1 |
| S | T | T | F | F | F |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| D | 0 | 34.83 | 29.07 | 34.06 | 14.21 |
| P | - | 0 | 4 | 4 | 1 |
| S | T | T | F | F | T |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| D | 0 | 34.83 | 29.07 | 34.06 | 14.21 |
| P | - | 0 | 4 | 4 | 1 |
| S | T | T | T | F | T |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| D | 0 | 34.83 | 29.07 | 34.06 | 14.21 |
| P | - | 0 | 4 | 4 | 1 |
| S | T | T | T | T | T |

Edges = {(0,1), (1,4), (4,2), (4,3)}

DFS = 0, 1, 4, 2, 3

Tour = 0, 1, 4, 2, 3, 0

Distance = 119.28

(This approximately matches the mst solution)

**Results of Instructor Test Cases:**

**Instructor Test Case #1:**

Optimal Tour: 5 Cities

Min Cost 179.30334267593292

0 1 4 3 2 0

A B E D C A

**Instructor Test Case #2:**

Optimal Tour: 12 Cities

Min Cost 580.2353924073391

0 10 8 1 9 4 3 6 11 7 5 2 0

A K I B J E D G L H F C A

**Instructor Test Case #1:**

Best Greedy Tour: 5 Cities

Min Cost 190.58549704246406

0 1 4 2 3 0

A B E C D A

**Instructor Test Case #2:**

Best Greedy Tour: 12 Cities

Min Cost 872.8222916142839

0 11 9 1 10 8 6 7 5 2 3 4 0

A L J B K I G H F C D E A

**Instructor Test Case #1:**

MST Tour: 5 Cities

Min Cost 119.30436349367712

0 1 4 2 3 0

A B E C D A

**Instructor Test Case #2:**

MST Tour: 12 Cities

Min Cost 628.7697297775677

0 2 10 8 3 4 6 11 7 5 1 9 0

A C K I D E G L H F B J A

/\*

\* Programmer: David Kopp

\* Project #: Project 2

\* File Name: TestETSP.java

\* Date: 10-25-15

\* Class: Comp496ALG

\* Description: This class is the main driver for the BFS, GS, and MSTS.

\* It times the method runs as well.

\*/

// Eclipse project package (comment out if not using eclipse project manager)

package project2;

// Import Statements

import java.io.FileNotFoundException;

// Class for Main Driver to run BFS, GS, and MSTS

public class TestETSP

{

public static void main(String args[]) throws FileNotFoundException

{

long startTime;

long stopTime;

long elapsedTime;

/\* Run these test cases 1 at a time and not all together. If you run them in sequence

\*\* you will get bad estimates because of JVM optimization and CPU caching.

\*\* (First of every group of the same objects take longer than subsequent objects of

\*\* the same type, thus you have to run them 1 at a time or in an alternating pattern.)

\*/

startTime = System.currentTimeMillis();

ETSPBruteForce bf1 = new ETSPBruteForce("sampleCase1.txt");

System.out.println(bf1.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPBruteForce bf4 = new ETSPBruteForce("sampleCase2.txt");

System.out.println(bf4.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPBruteForce bf2 = new ETSPBruteForce("instructorCase1.txt");

System.out.println(bf2.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPBruteForce bf3 = new ETSPBruteForce("instructorCase2.txt");

System.out.println(bf3.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPGreedy g1 = new ETSPGreedy("sampleCase1.txt", 0);

System.out.println(g1.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPGreedy g4 = new ETSPGreedy("sampleCase2.txt", 0);

System.out.println(g4.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPGreedy g5 = new ETSPGreedy("sampleCase3.txt", 0);

System.out.println(g5.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPGreedy g2 = new ETSPGreedy("instructorCase1.txt", 0);

System.out.println(g2.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPGreedy g3 = new ETSPGreedy("instructorCase2.txt", 0);

System.out.println(g3.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPMinSpanTree p1 = new ETSPMinSpanTree("sampleCase1.txt", 0);

System.out.println(p1.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPMinSpanTree p4 = new ETSPMinSpanTree("sampleCase2.txt", 0);

System.out.println(p4.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPMinSpanTree p5 = new ETSPMinSpanTree("sampleCase3.txt", 0);

System.out.println(p5.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPMinSpanTree p2 = new ETSPMinSpanTree("instructorCase1.txt", 0);

System.out.println(p2.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

startTime = System.currentTimeMillis();

ETSPMinSpanTree p3 = new ETSPMinSpanTree("instructorCase2.txt", 0);

System.out.println(p3.toString());

stopTime = System.currentTimeMillis();

elapsedTime = stopTime - startTime;

System.out.print("Run Time was " + elapsedTime +"msecs\n\n");

}

}

/\*

\* Programmer: David Kopp

\* Project #: Project 2

\* File Name: ETSPBruteForce.java

\* Date: 10-25-15

\* Class: Comp496ALG

\* Description: This class using brute force to solve the Euclidean Traveling Salesman Problem.

\*/

// Eclipse project package (comment out if not using eclipse project manager)

package project2;

// Import Statements

import java.io.File;

import java.util.Scanner;

import java.io.FileNotFoundException;

// Class for ETSP Brute Force

public class ETSPBruteForce

{

// Declared Variables

private int numberOfCities;

private double distanceOfPath;

// Input File Object

File file;

// Data Structures

private int[] pathTraveled;

private int[] permutationArray;

private double[] xloc;

private double[] yloc;

private double[][] distanceTable;

private String[] cityNames; //cityName[k] is name of city k

// Constructor for ETSPBruteForce

public ETSPBruteForce(String inputFile)

{

file = new File(inputFile);

cityNames = new String[26];

initArray(cityNames);

readInputData();

calculateDistances();

checkDistance(permutationArray);

permutations(permutationArray);

}

// Distance formula for two points in a Euclidean plane

public double distance(double x1, double x2, double y1, double y2)

{

return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));

}

// This method computes the distance traveled in an int[]

public double computeDistance(int[] traveledPath)

{

double totalDistance = 0;

//printArr(traveledPath);

for (int i = 0; i + 1 <= numberOfCities; i++)

{

totalDistance += distanceTable[traveledPath[i]][traveledPath[i + 1]];

//System.out.println("subtotal[" +i +"]: " +distanceTable[traveledPath[i]][traveledPath[i+1]]);

}

//System.out.println("Total: " +totalDistance);

return totalDistance;

}

// Returns a String of the pathTraveled array

public String printPath()

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i <= numberOfCities; i++)

{

sb.append(pathTraveled[i] + " ");

}

return sb.toString();

}

public void printArr(int[] arr)

{

for(int i = 0; i < arr.length; i++)

{

System.out.print(+ arr[i] + " ");

}

System.out.println("\n");

}

public String printArr(String[] arr)

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i < pathTraveled.length; i++)

{

sb.append(cityNames[pathTraveled[i]] + " ");

}

return sb.toString();

}

// Returns a String to print the information about the ETSPBruteForce object

public String toString()

{

return "Optimal Tour: " + numberOfCities + " Cities\nMin Cost " + distanceOfPath

+ "\n" + printPath() + "\n" + printArr(cityNames) + "\n";

}

// Private Methods

// Reads the input of the text file and initializes variables and data structures

private void readInputData()

{

try

{

Scanner input = new Scanner(file);

numberOfCities = input.nextInt();

pathTraveled = new int[numberOfCities + 1];

permutationArray = new int[numberOfCities];

xloc = new double[numberOfCities];

yloc = new double[numberOfCities];

distanceTable = new double[numberOfCities][numberOfCities];

initArray(pathTraveled);

initPermutationArray(permutationArray);

for (int i = 0; i < numberOfCities; i++)

{

xloc[i] = input.nextDouble();

}

for (int i = 0; i < numberOfCities; i++)

{

yloc[i] = input.nextDouble();

}

input.close();

} // End of Try

catch (FileNotFoundException e)

{

System.out.println(e);

System.exit(1);

} // End of Catch

}

// Calculates the distance between every point to every point on the Euclidean plane

private void calculateDistances()

{

for (int i = 0; i < numberOfCities; i++)

{

for (int j = 0; j < numberOfCities; j++)

{

distanceTable[i][j] = distance(xloc[i], xloc[j], yloc[i], yloc[j]);

}

}

}

/\* Got this algorithm from

\*\* http://stackoverflow.com/questions/2799078/permutation-algorithm-without-recursion-java

\*\* Posted by higuaro (Methods swap, reverse, and permutations)

\*/

private int factorial(int n) {

int fact = 1;

for (int i = 1; i <= n; i++) {

fact \*= i;

}

return fact;

}

// Swaps elements in an array

private void swap(int[] elements, int i, int j) {

int temp = elements[i];

elements[i] = elements[j];

elements[j] = temp;

}

// Reverses the elements of an array (in place) from the start index to the end index

private void reverse(int[] array, int startIndex, int endIndex) {

int size = endIndex + 1 - startIndex;

int limit = startIndex + size / 2;

for (int i = startIndex; i < limit; i++) {

// swap(array, i, startIndex + (size - 1 - (i - startIndex)));

swap(array, i, 2 \* startIndex + size - 1 - i);

}

}

/\*\*

\* Implements the Knuth's L-Algorithm permutation algorithm

\* modifying the collection in place

\*/

private void permutations(int[] sequence) {

final int N = sequence.length;

// There are n! permutations, but the first permutation is the array without

// modifications, so the number of permutations is n! - 1

int numPermutations = factorial(N) - 1;

// For every possible permutation

for (int n = 0; n < numPermutations; n++) {

// Iterate the array from right to left in search

// of the first couple of elements that are in ascending order

for (int i = N - 1; i >= 1; i--) {

// If the elements i and i - 1 are in ascending order

if (sequence[i - 1] < sequence[i]) {

// Then the index "i - 1" becomes our pivot index

int pivotIndex = i - 1;

// Scan the elements at the right of the pivot (again, from right to left)

// in search of the first element that is bigger

// than the pivot and, if found, swap it

for (int j = N - 1; j > pivotIndex; j--) {

if (sequence[j] > sequence[pivotIndex]) {

swap(sequence, j, pivotIndex);

break;

}

}

// Now reverse the elements from the right of the pivot index

// (this nice touch to the algorithm avoids the recursion)

reverse(sequence, pivotIndex + 1, N - 1);

break;

}

}

checkDistance(sequence);

}

}

/\* Checks the distance between the pathTraveled array and the passed next permutation.

\* If the permutation is shorter in distance then it becomes pathTraveled.

\* Otherwise nothing happens.

\*/

private void checkDistance(int[] pathGenerated)

{

if (pathTraveled[0] == -1)

{

pathTraveled = copyArray(pathGenerated);

distanceOfPath = computeDistance(pathTraveled);

}

else

{

int[] tempArr = new int[numberOfCities + 1];

tempArr = copyArray(pathGenerated);

double generatedPathDistance = computeDistance(tempArr);

if (distanceOfPath > generatedPathDistance)

{

pathTraveled = tempArr;

distanceOfPath = generatedPathDistance;

}

}

}

// Initiates an array to -1

private void initArray(int[] inputArray)

{

int length = inputArray.length;

for (int i = 0; i < length; i++)

{

inputArray[i] = -1;

}

}

private void initArray(String[] arr)

{

char[] chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();

for (int i = 0; i < arr.length; i++)

{

arr[i] = Character.toString(chars[i]);

}

}

// Initiates the permutationArray to 0 to n-1

private void initPermutationArray(int[] inputArray)

{

int length = inputArray.length;

for (int i = 0; i < length; i++)

{

inputArray[i] = i;

}

}

// This method copies the array and adds another slot for the first city

private int[] copyArray(int[] arr1)

{

int[] newArr = new int[numberOfCities + 1];

for (int i = 0; i < arr1.length; i++)

{

newArr[i] = arr1[i];

}

newArr[numberOfCities] = arr1[0];

return newArr;

}

}

/\*

\* Programmer: David Kopp

\* Project #: Project 2

\* File Name: ETSPGreedy.java

\* Date: 10-25-15

\* Class: Comp496ALG

\* Description: This class using the Greedy Solution to solve

\* the Euclidean Traveling Salesman Problem.

\*/

// Eclipse project package (comment out if not using eclipse project manager)

package project2;

// Import Statements

import java.io.File;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.Scanner;

// Class for Greedy Solution

public class ETSPGreedy

{

// Declared Variables

private int numberOfCities;

private double distanceOfPath;

// Input File Object

File file;

// Data Structures

private ArrayList<Integer> pathTraveled;

private boolean[] cityVisited;

private double[] xloc;

private double[] yloc;

private double[][] distanceTable;

private String[] cityNames; //cityName[k] is name of city k

// Constructor for ETSPGreedy

public ETSPGreedy(String inputFile, int startCity)

{

file = new File(inputFile);

cityNames = new String[26];

initArray(cityNames);

readInputData();

calculateDistances();

//printDistanceTable();

greedyTour(startCity);

pathTraveled.add(pathTraveled.get(0));

distanceOfPath = computeDistance(pathTraveled);

}

// Prints the distance table for troubleshooting

public void printDistanceTable()

{

for (int i = 0; i < numberOfCities; i++)

{

for (int j = 0; j < numberOfCities; j++)

{

System.out.print(distanceTable[i][j] + " ");

}

System.out.print("\n");

}

}

// This method is the Greedy Algorithm for the ETSP problem

private void greedyTour(int startCity)

{

pathTraveled.add(startCity);

cityVisited[startCity] = true;

int path = nextCity(startCity);

while (!hasVisitedAll())

{

pathTraveled.add(path);

cityVisited[path] = true;

path = nextCity(path);

}

}

// This method finds the next closest unvisited city

public int nextCity(int fromCity)

{

int toCity = firstNotVisited();

double temp1 = distanceTable[fromCity][toCity];

double temp2 = temp1;

for(int i = toCity; i < numberOfCities; i++)

{

if (!cityVisited[i])

{

temp1 = distanceTable[fromCity][i];

if (temp1 < temp2 && temp1 != 0 && temp2 != 0)

{

toCity = i;

}

}

}

return toCity;

}

// This method finds out if all the cities have been visited

public boolean hasVisitedAll()

{

boolean result = true;

for (int i = 0; i < cityVisited.length; i++)

{

if (!cityVisited[i])

{

result = false;

break;

}

}

return result;

}

// This method finds the first unvisited city

public int firstNotVisited()

{

int result = 0;

for (int i = 0; i < cityVisited.length; i++)

{

if (!cityVisited[i])

{

result = i;

break;

}

}

return result;

}

// Distance formula for two points in a Euclidean plane

public double distance(double x1, double x2, double y1, double y2)

{

return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));

}

// This method computes the distance traveled in an ArrayList<Integer>

public double computeDistance(ArrayList<Integer> traveledPath)

{

double totalDistance = 0;

//printArr(traveledPath);

for (int i = 0; i + 1 <= numberOfCities; i++)

{

totalDistance += distanceTable[traveledPath.get(i)][traveledPath.get(i + 1)];

//System.out.println("subtotal[" +i +"]: " +distanceTable[traveledPath[i]][traveledPath[i+1]]);

}

//System.out.println("Total: " +totalDistance);

return totalDistance;

}

// Returns a String of the pathTraveled array

public String printPath()

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i <= numberOfCities; i++)

{

sb.append(pathTraveled.get(i) + " ");

}

return sb.toString();

}

public void printArr(int[] arr)

{

for (int i = 0; i < arr.length; i++)

{

System.out.print(+ arr[i] + " ");

}

System.out.println("\n");

}

public String printArr(String[] arr)

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i < pathTraveled.size(); i++)

{

sb.append(cityNames[pathTraveled.get(i)] + " ");

}

return sb.toString();

}

// Returns a String to print the information about the ETSPBruteForce object

public String toString()

{

return "Best Greedy Tour: " + numberOfCities + " Cities\nMin Cost " + distanceOfPath

+ "\n" + printPath() + "\n" + printArr(cityNames) + "\n";

}

// Private Methods

// Reads the input of the text file and initializes variables and data structures

private void readInputData()

{

try

{

Scanner input = new Scanner(file);

numberOfCities = input.nextInt();

cityVisited = new boolean[numberOfCities];

pathTraveled = new ArrayList<Integer>();

xloc = new double[numberOfCities];

yloc = new double[numberOfCities];

distanceTable = new double[numberOfCities][numberOfCities];

initArray(cityVisited);

for (int i = 0; i < numberOfCities; i++)

{

xloc[i] = input.nextDouble();

}

for (int i = 0; i < numberOfCities; i++)

{

yloc[i] = input.nextDouble();

}

input.close();

} // End of Try

catch (FileNotFoundException e)

{

System.out.println(e);

System.exit(1);

} // End of Catch

}

// Calculates the distance between every point to every point on the Euclidean plane

private void calculateDistances()

{

for (int i = 0; i < numberOfCities; i++)

{

for (int j = 0; j < numberOfCities; j++)

{

distanceTable[i][j] = distance(xloc[i], xloc[j], yloc[i], yloc[j]);

}

}

}

// Initiates the cityVisited array

private void initArray(boolean[] arr)

{

int length = arr.length;

for (int i = 0; i < length; i++)

{

arr[i] = false;

}

}

// Initiates the cityName array

private void initArray(String[] arr)

{

char[] chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();

for (int i = 0; i < arr.length; i++)

{

arr[i] = Character.toString(chars[i]);

}

}

}

/\*

\* Programmer: David Kopp

\* Project #: Project 2

\* File Name: ETSPMinSpanTree.java

\* Date: 10-25-15

\* Class: Comp496ALG

\* Description: This class using Prim's Algorithm and a depth first search

\* solution to solve the Euclidean Traveling Salesman Problem.

\*/

// Eclipse project package (comment out if not using eclipse project manager)

package project2;

// Import Statements

import java.io.File;

import java.io.FileNotFoundException;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.Scanner;

import java.util.Stack;

// Class for Prim's Algorithm Solution

public class ETSPMinSpanTree

{

// Declared Variables

private int numberOfCities;

private double distanceOfPath;

// Input File Object

File file;

// Data Structures

private ArrayList<Integer> pathTraveled;

private double[] xloc;

private double[] yloc;

private double[][] distanceTable;

private double[][] graphT;

private boolean[] cityVisited;

private int[] parent;

private ArrayList<Integer> s = new ArrayList<Integer>();

private double[] distance;

private String[] cityNames; //cityName[k] is name of city k

// Constructor for ETSPMinSpanTree

public ETSPMinSpanTree(String inputFile, int startCity)

{

file = new File(inputFile);

cityNames = new String[26];

initArray(cityNames);

readInputData();

calculateDistances();

//printTable(distanceTable);

primsTour(startCity);

//printEdges();

createGraphT(startCity);

pathTraveled.add(pathTraveled.get(0));

distanceOfPath = computeDistance(pathTraveled);

}

// Prints the edges of prim's algorithm

public void printEdges()

{

for (int i = 1; i < numberOfCities; i++)

{

System.out.println("Edge " + i + ": (" + parent[s.get(i)] + ", "

+ s.get(i) + ") " + distance[s.get(i)]);

}

}

// Prim's Algorithm generates a MST

private void primsTour(int startCity)

{

int currentParent = startCity;

distance[currentParent] = 0;

cityVisited[currentParent] = true;

parentOf(currentParent);

s.add(currentParent);

for (int i = 1; i < numberOfCities; i++)

{

int u = findSmallestU(i);

s.add(u);

for (int j = 0; j < numberOfCities; j++)

{

if (!s.contains(j) && distanceTable[u][j] != 0)

{

if (distanceTable[u][j] < distance[j])

{

distance[j] = distanceTable[u][j];

parent[j] = u;

}

}

}

cityVisited[u] = true;

}

}

// Creates the graphT and does a dfs on it.

private void createGraphT(int startCity)

{

for (int i = 1; i < numberOfCities; i++)

{

graphT[parent[s.get(i)]][s.get(i)] = distance[s.get(i)];

graphT[s.get(i)][parent[s.get(i)]] = distance[s.get(i)];

}

//printTable(graphT);

initArray(cityVisited);

dfs(startCity);

//Collections.reverse(pathTraveled);

//System.out.println(pathTraveled.toString());

}

// Recursion method for DFS Traversal

private void dfs(int start)

{

Stack<Integer> stack = new Stack<Integer>();

boolean[] visited = new boolean[numberOfCities];

initArray(visited);

stack.push(start);

visited[start] = true;

pathTraveled.add(start);

//System.out.print(start + " ");

while (!stack.isEmpty())

{

int stackTop = stack.peek();

for (int i = start; i < numberOfCities; i++)

{

if (graphT[stackTop][i] != 0 && !visited[i])

{

stack.push(i);

visited[i] = true;

//System.out.print(i + " ");

stackTop = i;

pathTraveled.add(stackTop);

i = 1;

}

}

stack.pop();

}

}

// Finds the smallest distance for u

public int findSmallestU(int u)

{

double smallestDistance = Double.POSITIVE\_INFINITY;

int result = 1;

for (int i = 0; i < numberOfCities; i++)

{

if (distance[i] < smallestDistance && !cityVisited[i])

{

smallestDistance = distance[i];

result = i;

}

}

return result;

}

// Inserts the parent city to all children cities

private void parentOf(int parent)

{

for (int i = 0; i < numberOfCities; i++)

{

if (distanceTable[parent][i] != 0 && !cityVisited[i])

{

distance[i] = distanceTable[parent][i];

this.parent[i] = parent;

}

}

}

// Finds the next city to visit

public int nextVisited(int city)

{

int result = -1;

for (int i = 0; i < numberOfCities; i++)

{

if (cityVisited[i] && i != city)

{

result = i;

break;

}

}

return result;

}

// This method finds the next closest unvisited city

public int nextCity(int fromCity)

{

int toCity = firstNotVisited();

double temp1 = distanceTable[fromCity][toCity];

double temp2 = temp1;

for(int i = toCity; i < numberOfCities; i++)

{

if (!cityVisited[i])

{

temp1 = distanceTable[fromCity][i];

}

if (temp1 < temp2 && temp1 != 0 && temp2 != 0)

{

toCity = i;

}

else if (temp1 == temp2)

{

temp2 = temp1;

}

}

return toCity;

}

// This method finds out if all the cities have been visited

public boolean hasVisitedAll()

{

boolean result = true;

for (int i = 0; i < cityVisited.length; i++)

{

if (!cityVisited[i])

{

result = false;

break;

}

}

return result;

}

// This method finds the first unvisited city

public int firstNotVisited()

{

int result = 0;

for (int i = 0; i < cityVisited.length; i++)

{

if (!cityVisited[i])

{

result = i;

break;

}

}

return result;

}

// Prints the distance table for troubleshooting

public void printTable(double[][] arr)

{

for (int i = 0; i < numberOfCities; i++)

{

for (int j = 0; j < numberOfCities; j++)

{

System.out.print(arr[i][j] + " ");

}

System.out.print("\n");

}

}

// Distance formula for two points in a Euclidean plane

public double distance(double x1, double x2, double y1, double y2)

{

return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));

}

// This method computes the distance traveled in an ArrayList<Integer>

public double computeDistance(ArrayList<Integer> traveledPath)

{

double totalDistance = 0;

//printArr(traveledPath);

for (int i = 0; i + 1 < numberOfCities; i++)

{

totalDistance += distanceTable[traveledPath.get(i)][traveledPath.get(i + 1)];

//System.out.println("subtotal[" +i +"]: " +distanceTable[traveledPath[i]][traveledPath[i+1]]);

}

//System.out.println("Total: " +totalDistance);

return totalDistance;

}

// Returns a String of the pathTraveled array

public String printPath()

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i < numberOfCities; i++)

{

sb.append(pathTraveled.get(i) + " ");

}

return sb.toString();

}

// Prints out an int[]

public void printArr(int[] arr)

{

for (int i = 0; i < arr.length; i++)

{

System.out.print(+ arr[i] + " ");

}

System.out.println("\n");

}

// Labels for all the cities

public String printArr(String[] arr)

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i < pathTraveled.size(); i++)

{

sb.append(cityNames[pathTraveled.get(i)] + " ");

}

return sb.toString();

}

// Returns a String to print the information about the ETSPBruteForce object

public String toString()

{

return "MST Tour: " + numberOfCities + " Cities\nMin Cost " + distanceOfPath

+ "\n" + printPath() + "\n" + printArr(cityNames) + "\n";

}

// Private Methods

// Reads the input of the text file and initializes variables and data structures

private void readInputData()

{

try

{

Scanner input = new Scanner(file);

numberOfCities = input.nextInt();

cityVisited = new boolean[numberOfCities];

pathTraveled = new ArrayList<Integer>();

parent = new int[numberOfCities];

distance = new double[numberOfCities];

xloc = new double[numberOfCities];

yloc = new double[numberOfCities];

distanceTable = new double[numberOfCities][numberOfCities];

graphT = new double[numberOfCities][numberOfCities];

initArray(cityVisited);

initArray(parent);

initArray(distance);

initArray(graphT);

for (int i = 0; i < numberOfCities; i++)

{

xloc[i] = input.nextDouble();

}

for (int i = 0; i < numberOfCities; i++)

{

yloc[i] = input.nextDouble();

}

input.close();

} // End of Try

catch (FileNotFoundException e)

{

System.out.println(e);

System.exit(1);

} // End of Catch

}

// Calculates the distance between every point to every point on the Euclidean plane

private void calculateDistances()

{

for (int i = 0; i < numberOfCities; i++)

{

for (int j = 0; j < numberOfCities; j++)

{

distanceTable[i][j] = distance(xloc[i], xloc[j], yloc[i], yloc[j]);

}

}

}

// Initiates the parent array

private void initArray(int[] arr)

{

int length = arr.length;

for (int i = 0; i < length; i++)

{

arr[i] = -1;

}

}

// Initiates the distance array

private void initArray(double[] arr)

{

int length = arr.length;

for (int i = 0; i < length; i++)

{

arr[i] = Double.POSITIVE\_INFINITY;

}

}

// Initiates the cityVisited array

private void initArray(boolean[] arr)

{

int length = arr.length;

for (int i = 0; i < length; i++)

{

arr[i] = false;

}

}

// Initiates the graphT array[][]

private void initArray(double[][] arr)

{

for (int i = 0; i < arr.length; i++)

{

for (int j = 0; j < arr.length; j++)

{

arr[i][j] = 0;

}

}

}

// Initiates the cityName array

private void initArray(String[] arr)

{

char[] chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();

for (int i = 0; i < arr.length; i++)

{

arr[i] = Character.toString(chars[i]);

}

}

}