Лабораторна робота №9

Виконав: студент 1-го курсу ФІОТ групи ІП-92 Медведєв Михайло Євгенович

**Завдання:**

Реалізувати програмне застосування (програму), яке виконує наступні функції.

1.Зчитування данихз вхідного файлу. На вхід подається текстовий файл наступного вигляду:

N

x1y1

x2y2

....

Xnyn

Тут n–кількість вершин графу (ціле число, більше нуля), xiта yi–координати вершини на декартовій площині. Відстані між вершинами обраховуються як відстані по прямій між двома точками.

2.Знайти розв’язок задачі комівояжера для заданого графу. Програма виводить знайдений гамільтонів цикл найменшої вартості.

Код:

**package** com.company;  
  
**import** java.io.File;  
**import** java.io.FileNotFoundException;  
**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.Scanner;  
  
**public class** DS\_IP92\_LR9\_MedvedievM {  
 **public static void** main(String[] args) **throws** FileNotFoundException {  
 Graph graph = **new** Graph(**new** File("inputs/input.txt"));  
 graph.findMinimumWay();  
 System.*out*.println("Minimal cost: " + graph.minimalCost);  
 System.*out*.print("Way: ");  
 graph.printWay(graph.bestWay);  
 }  
  
}  
  
**class** Graph {  
 City[] cities;  
 **double**[][] distanceMatrix;  
  
 Graph(File file) **throws** FileNotFoundException {  
 getCities(file);  
 setDistanceMatrix();  
 //testing  
// distanceMatrix = new double[][]{  
// {-1, 20, 18, 12, 8},  
// {5, -1, 14, 7, 11},  
// {12, 18, -1, 6, 11},  
// {11, 17, 11, -1, 12},  
// {5, 5, 5, 5, -1}};  
 }  
  
 **protected** String matrixToString(**double**[][] matrix, String extraText) {  
 StringBuilder outputText = **new** StringBuilder(extraText + "\n");  
  
 **for** (**int** i = 0; i < matrix.length; i++) {  
 **for** (**int** j = 0; j < matrix[0].length; j++)  
 outputText.append((matrix[i][j] >= 0) ? " " : "").append(matrix[i][j]).append(" ");  
  
 outputText.append("\n");  
 }  
 **return** outputText.toString();  
 }  
  
 **private void** getCities(File file) **throws** FileNotFoundException {  
 Scanner scanner = **new** Scanner(file);  
 **int** numberOfCities = Integer.*parseInt*(scanner.nextLine());  
 cities = **new** City[numberOfCities];  
 **for** (**int** i = 0; i < numberOfCities; i++) {  
 String line = scanner.nextLine();  
 String[] numbers = line.split(" ");  
 cities[i] = **new** City(Integer.*parseInt*(numbers[0]), Integer.*parseInt*(numbers[1]));  
 }  
 }  
  
 **private void** setDistanceMatrix() {  
 distanceMatrix = **new double**[cities.length][cities.length];  
 **for** (**int** i = 0; i < cities.length; i++) {  
 **for** (**int** j = i + 1; j < cities.length; j++) {  
 **double** distance = cities[i].distanceToCity(cities[j]);  
 distanceMatrix[i][j] = distance;  
 distanceMatrix[j][i] = distance;  
 }  
 }  
 **for** (**int** i = 0; i < cities.length; i++) {  
 distanceMatrix[i][i] = -1;  
 }  
  
// System.out.println(matrixToString(distanceMatrix, ""));  
 }  
  
 **double** minimalCost = Integer.*MAX\_VALUE*;  
 ArrayList<**int**[]> bestWay;  
  
 **public void** findMinimumWay() {  
 ArrayList<**int**[]> way = **new** ArrayList<>();  
 **boolean**[] doneLines = **new boolean**[distanceMatrix.length], doneColumns = **new boolean**[distanceMatrix.length];  
 **double**[][] startDistanceMatrix = getCopyOfMatrix(**this**.distanceMatrix);  
 findRecurs(startDistanceMatrix, 0, doneLines, doneColumns, way);  
  
 bestWay = sortWay(bestWay);  
  
 }  
  
 **private** ArrayList<**int**[]> sortWay(ArrayList<**int**[]> way) {  
 **int** startLength = way.size();  
 ArrayList<**int**[]> output = **new** ArrayList<>();  
 **int**[] currentVerge = way.remove(0);  
 output.add(currentVerge);  
 **while** (output.size() != startLength) {  
 currentVerge = output.get(output.size() - 1);  
 **int** lastIndex = currentVerge[1];  
 **for** (**int** i = 0; i < way.size(); i++) {  
 **if** (way.get(i)[0] == lastIndex) {  
 output.add(way.remove(i));  
 **break**;  
 }  
 }  
 }  
  
 way = output;  
 **return** way;  
 }  
  
  
 **private void** findRecurs(**double**[][] distanceMatrix, **double** minLimit, **boolean**[] doneLines, **boolean**[] doneColumns, ArrayList<**int**[]> way) {  
  
 minLimit += reduce(distanceMatrix, doneLines, doneColumns);  
  
// printWay(way);  
 **if** (minLimit > minimalCost) {  
  
 **return**;  
  
 }  
  
 **double**[] currentWay = findZeros(distanceMatrix, doneLines, doneColumns);  
  
 **double**[][] matrix2 = getCopyOfMatrix(distanceMatrix);  
// System.out.println(currentWay[0] + " " + currentWay[1]);  
 **if** (currentWay[0] == -1 || (**int**) currentWay[0] == (**int**) currentWay[1])  
 **return**;  
  
// System.out.println("K");  
 matrix2[(**int**) currentWay[0]][(**int**) currentWay[1]] = -1;  
  
 **double**[][] matrix1 = distanceMatrix;  
 matrix1[(**int**) currentWay[1]][(**int**) currentWay[0]] = -1;  
// addInfinity(matrix1);  
 **boolean**[] doneLines1 = doneLines.clone();  
 doneLines1[(**int**) currentWay[0]] = **true**;  
 **boolean**[] doneColumns1 = doneColumns.clone();  
 doneColumns1[(**int**) currentWay[1]] = **true**;  
 ArrayList<**int**[]> way1 = **new** ArrayList<>();  
 way1.addAll(way);  
 way1.add(**new int**[]{(**int**) currentWay[0], (**int**) currentWay[1]});  
 **double** minLimit1 = minLimit;  
  
// printWay(way1);  
 **if** (way1.size() >= distanceMatrix.length - 2) {  
// System.out.println(matrixToString(distanceMatrix, "RESULT"));  
// printWay(way1);  
// System.out.println(Arrays.toString(doneLines1));  
// System.out.println(Arrays.toString(doneColumns1));  
 **double** sum = 0;  
  
 **int** line1 = -1, line2 = -1, column1 = -1, column2 = -1;  
 **for** (**int** i = 0; i < doneLines1.length; i++) {  
 **if** (!doneLines1[i]) {  
 **if** (line1 == -1)  
 line1 = i;  
 **else** {  
 line2 = i;  
 **break**;  
 }  
 }  
 }  
 **for** (**int** i = doneColumns1.length - 1; i >= 0; i--) {  
 **if** (!doneColumns1[i]) {  
 **if** (column1 == -1)  
 column1 = i;  
 **else** {  
 column2 = i;  
 **break**;  
 }  
 }  
 }  
  
 **if** (line1 == column1 || line2 == column2)  
 **return**;  
  
 way1.add(**new int**[]{line1, column1});  
 way1.add(**new int**[]{line2, column2});  
 **for** (**int**[] verge : way1) {  
 sum += **this**.distanceMatrix[verge[0]][verge[1]];  
 }  
  
 **if** (sum < minimalCost) {  
 minimalCost = sum;  
 bestWay = way1;  
 }  
 **return**;  
 }  
  
  
 findRecurs(matrix1, minLimit1, doneLines1, doneColumns1, way1);  
 findRecurs(matrix2, minLimit, doneLines, doneColumns, way);  
  
  
 }  
  
 **public void** printWay(ArrayList<**int**[]> way) {  
 **for** (**int** i = 0; i < way.size(); i++) {  
 System.*out*.print("[" + way.get(i)[0] + ", " + way.get(i)[1] + "] ");  
 }  
 System.*out*.println();  
 }  
  
 **void** addInfinity(**double**[][] matrix) {  
  
 **boolean**[] infRow = **new boolean**[matrix[0].length], infColumn = **new boolean**[matrix.length];  
  
 **for** (**int** i = 0; i < matrix.length; i++) {  
 **for** (**int** j = 0; j < matrix[0].length; j++) {  
 **if** (matrix[i][j] == -1) {  
 infRow[i] = **true**;  
 infColumn[j] = **true**;  
 }  
 }  
 }  
  
 **int** notInf = 0;  
 **for** (**int** i = 0; i < infRow.length; i++)  
 **if** (!infRow[i]) {  
 notInf = i;  
 **break**;  
 }  
  
  
 **for** (**int** j = 0; j < infColumn.length; j++)  
 **if** (!infColumn[j]) {  
 matrix[notInf][j] = -1;  
 **break**;  
 }  
  
 }  
  
  
 **double**[][] getCopyOfMatrix(**double**[][] matrix) {  
 **double**[][] output = **new double**[matrix.length][matrix[0].length];  
 **for** (**int** i = 0; i < matrix.length; i++) {  
 **for** (**int** j = 0; j < matrix[0].length; j++) {  
 output[i][j] = matrix[i][j];  
 }  
 }  
 **return** output;  
 }  
  
 **private double**[] findZeros(**double**[][] distanceMatrix, **boolean**[] doneLines, **boolean**[] doneColumns) {  
 **int** indexI = -1, indexJ = -1;  
// System.out.println(matrixToString(distanceMatrix, "KUKU"));  
// System.out.println(Arrays.toString(doneLines));  
// System.out.println(Arrays.toString(doneColumns));  
 **double** maxValue = -1;  
 **for** (**int** i = 0; i < distanceMatrix.length; i++) {  
 **if** (doneLines[i])  
 **continue**;  
  
 **for** (**int** j = 0; j < distanceMatrix[0].length; j++) {  
 **if** (!doneColumns[j] && distanceMatrix[i][j] == 0) {  
 **double** fine = findFineForZero(distanceMatrix, i, j, doneLines, doneColumns);  
// System.out.println(fine);  
 **if** (fine > maxValue) {  
 maxValue = fine;  
 indexI = i;  
 indexJ = j;  
 }  
 }  
 }  
 }  
  
 **return new double**[]{indexI, indexJ, maxValue};  
 }  
  
 **private double** findFineForZero(**double**[][] distanceMatrix, **int** i, **int** j, **boolean**[] doneLines, **boolean**[] doneColumns) {  
 **double** minLine = Integer.*MAX\_VALUE*, minColumn = Integer.*MAX\_VALUE*;  
 **for** (**int** k = 0; k < distanceMatrix[0].length; k++) {  
 **if** (k == j || doneColumns[k] || distanceMatrix[i][k] == -1)  
 **continue**;  
// System.out.println("D: " + distanceMatrix[i][k]);  
 **if** (distanceMatrix[i][k] < minLine)  
 minLine = distanceMatrix[i][k];  
 }  
  
 **for** (**int** k = 0; k < distanceMatrix.length; k++) {  
 **if** (k == i || doneLines[k] || distanceMatrix[k][j] == -1)  
 **continue**;  
  
 **if** (distanceMatrix[k][j] < minColumn)  
 minColumn = distanceMatrix[k][j];  
 }  
 **return** minLine + minColumn;  
 }  
  
 **private double** reduce(**double**[][] distanceMatrix, **boolean**[] doneLines, **boolean**[] doneColumns) {  
 **double** sumLine = reduceLines(distanceMatrix, doneLines);  
 **double** sumColumn = reduceColumns(distanceMatrix, doneColumns);  
 **return** sumLine + sumColumn;  
 }  
  
  
 **private double** reduceLines(**double**[][] distanceMatrix, **boolean**[] doneLines) {  
 **double**[] minimums = **new double**[distanceMatrix.length];  
  
 **for** (**int** i = 0; i < distanceMatrix.length; i++) {  
 **if** (doneLines[i])  
 **continue**;  
  
 **double** minimum = Integer.*MAX\_VALUE*;  
 **for** (**int** j = 0; j < distanceMatrix[0].length; j++) {  
 **if** (distanceMatrix[i][j] != -1 && distanceMatrix[i][j] < minimum) {  
 minimum = distanceMatrix[i][j];  
 }  
 }  
 **for** (**int** j = 0; j < distanceMatrix[0].length; j++) {  
 **if** (distanceMatrix[i][j] != -1) {  
 distanceMatrix[i][j] -= minimum;  
 }  
 }  
 minimums[i] = minimum;  
 }  
 **double** result = 0;  
 **for** (**int** i = 0; i < minimums.length; i++) {  
 result += minimums[i];  
 }  
 **return** result;  
 }  
  
 **private double** reduceColumns(**double**[][] distanceMatrix, **boolean**[] doneColumns) {  
 **double**[] minimums = **new double**[distanceMatrix.length];  
  
 **for** (**int** i = 0; i < distanceMatrix[0].length; i++) {  
 **if** (doneColumns[i])  
 **continue**;  
  
 **double** minimum = Integer.*MAX\_VALUE*;  
 **for** (**int** j = 0; j < distanceMatrix.length; j++) {  
 **if** (distanceMatrix[j][i] != -1 && distanceMatrix[j][i] < minimum) {  
 minimum = distanceMatrix[j][i];  
 }  
 }  
 **for** (**int** j = 0; j < distanceMatrix.length; j++) {  
 **if** (distanceMatrix[j][i] != -1) {  
 distanceMatrix[j][i] -= minimum;  
 }  
 }  
 minimums[i] = minimum;  
 }  
  
 **double** result = 0;  
 **for** (**int** i = 0; i < minimums.length; i++) {  
 result += minimums[i];  
 }  
 **return** result;  
 }  
}  
  
**class** City {  
 **int** x, y;  
  
 City(**int** x, **int** y) {  
 **this**.x = x;  
 **this**.y = y;  
 }  
  
 **double** distanceToCity(City nextCity) {  
 **return** Math.*sqrt*(Math.*pow*(**this**.x - nextCity.x, 2) + Math.*pow*(**this**.y - nextCity.y, 2));  
 }  
}

Результати роботи програми:



