Лабораторна робота №5.2

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

**Завдання:**

Реалізувати програмне застосування (програму), яке виконує наступні функції.При реалізації вважати, що заданий граф є орієнтованим.

Частина перша:

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

n m

v1u1w1

v2u2w2

...

....

Vmumwm

Тут n–кількість вершин графу (ціле число, більше нуля), m–кількість ребер графу (ціле число), viта ui–початкова та кінцева вершина ребра i(1≤vi≤n, 1≤ui≤n, цілі числа), wi–вага ребра (vi,ui). Індексація вершин у файлі ведеться з 1.

2.Визначити найкоротший маршрут між двома вершинами та його довжину.За допомогою алгоритму Белмана-Фордавизначити найкоротшу відстань між двома заданими вершинами (які вводяться користувачем), а також вивести сам знайдений найкоротший маршрут. Програма повинна коректно опрацьовувати факт наявностів графі циклу з від’ємною вагою.

3.Визначити найкоротшу відстань від заданої вершини до всіх інших вершин.За допомогою алгоритму Белмана-Форда визначити найкоротшу відстань від заданої вершини (вводиться користувачем) до всіх інших вершин графу. Програма виводить на екран список вершин із відповідними значеннями найкоротших відстаней.

Частина друга:

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

n m

v1u1w1

v2u2w2

...

....

Vmumwm

2.Визначити найкоротший маршрут між двома вершинами та його довжину.За допомогою алгоритму Джонсона визначити найкоротшу відстань між двома заданими вершинами (які вводяться користувачем), а також вивести сам знайдений найкоротший маршрут. Програма повинна коректно опрацьовувати факт наявностів графі циклу з від’ємною вагою.

3.Визначити найкоротшу відстань від заданої вершини до всіх інших вершин.За допомогою алгоритму Джонсона визначити найкоротшу відстань від заданої вершини (вводиться користувачем) до всіх інших вершин графу. Програма виводить на екран список вершин із відповідними значеннями найкоротших відстаней.

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\_LR52\_MedvedievM {  
 **public static void** main(String[] args) **throws** FileNotFoundException {  
 Graph graph = **new** Graph(**new** File("inputs/input.txt"));  
// graph.findShortestWaysJohnson(2);  
// graph.findShortestWayDijkstra(2,1);  
// graph.findShortestWayBetweenTwoNodesBellman(2,1);  
// graph.findShortestWaysBellman(1);  
// graph.findShortestWaysFloyd();  
// graph.findShortestWayBetweenTwoNodesBellman(2, 1);  
// graph.findShortestWayDijkstra(2, 1);  
 *doMenu*(graph);  
 }  
  
 **static void** doMenu(Graph graph) {  
 Scanner consoleScanner = **new** Scanner(System.*in*);  
 System.*out*.print("Bellman(1) or Johnson(2): ");  
 **int** choice = Integer.*parseInt*(consoleScanner.nextLine());  
  
  
 System.*out*.print("All distances(1) or only one way(2): ");  
 **int** choice2 = Integer.*parseInt*(consoleScanner.nextLine());  
 **if** (choice2 == 1) {  
 System.*out*.print("Input node: ");  
 **int** node = consoleScanner.nextInt();  
 **if** (choice == 1)  
 graph.findShortestWaysBellman(node);  
 **else if** (choice == 2)  
 graph.findShortestWaysJohnson(node);  
 } **else if** (choice2 == 2) {  
 System.*out*.print("Input two nodes: ");  
 String[] nodes = consoleScanner.nextLine().split(" ");  
 **if** (choice == 1)  
 graph.findShortestWayBellman(Integer.*parseInt*(nodes[0]), Integer.*parseInt*(nodes[1]));  
 **else if** (choice == 2)  
 graph.findShortestWayJohnson(Integer.*parseInt*(nodes[0]), Integer.*parseInt*(nodes[1]));  
  
  
 }  
  
 }  
}  
  
 **class** Graph {  
  
 **private int**[][] vertex;  
 **private int** numberOfNodes, numberOfVertex;  
 **private int**[][] adjacencyMatrix;  
 **private boolean** hasMinusVertex = **false**;  
 **private final int** MAX\_VALUE = Integer.*MAX\_VALUE* / 3;  
  
 Graph(File file) **throws** FileNotFoundException {  
 readFile(file);  
 **this**.adjacencyMatrix = getAdjacencyMatrix();  
 }  
  
 **public boolean** isHasMinusVertex() {  
 **return** hasMinusVertex;  
 }  
  
 **private void** readFile(File file) **throws** FileNotFoundException {  
 Scanner scanner = **new** Scanner(file);  
 **this**.numberOfNodes = scanner.nextInt();  
 **this**.numberOfVertex = scanner.nextInt();  
 **this**.vertex = **new int**[numberOfVertex][3];  
 **for** (**int** i = 0; i < numberOfVertex; i++) {  
 **for** (**int** j = 0; j < 3; j++) {  
 vertex[i][j] = scanner.nextInt();  
 }  
 **if** (!hasMinusVertex && vertex[i][2] < 0)  
 hasMinusVertex = **true**;  
 }  
 scanner.close();  
 }  
  
 **public void** findShortestWaysJohnson(**int** node) {  
 node--;  
 **int**[][] res = findArrayOfShortestWaysJohnson(node);  
 **if** (res.length == 0)  
 **return**;  
 **int**[] distances = res[0];  
 **for** (**int** i = 0; i < distances.length; i++) {  
 **if** (i != node)  
 System.*out*.println((node + 1) + "->" + (i + 1) + ": " + ((distances[i] == MAX\_VALUE) ? "∞" : distances[i]));  
 }  
 }  
  
 **public void** findShortestWayJohnson(**int** node1, **int** node2) {  
 node1--;  
 **int**[][] res = findArrayOfShortestWaysJohnson(node1);  
 **if** (res.length == 0)  
 **return**;  
 **int**[] distances = res[0];  
 **int**[] arrayP = res[1];  
 **if**(distances[node2-1] == MAX\_VALUE){  
 System.*out*.println("There\'s no way between these nodes!");  
 **return**;  
 }  
 **int**[] way = findWay(arrayP, node1, node2);  
 System.*out*.println("Minimal distance: " + distances[node2-1]);  
 System.*out*.println("Way: " + Arrays.*toString*(way));  
 }  
  
  
 **private int**[][] findArrayOfShortestWaysJohnson(**int** node) {  
  
 **int**[][] newVertex = **new int**[vertex.length + numberOfNodes][3];  
 **for** (**int** i = 0; i < vertex.length; i++) {  
 **for** (**int** j = 0; j < vertex[0].length; j++) {  
 newVertex[i][j] = vertex[i][j];  
 }  
 }  
 **for** (**int** i = vertex.length; i < vertex.length + numberOfNodes; i++) {  
 newVertex[i][0] = numberOfNodes + 1;  
 newVertex[i][1] = i - vertex.length + 1;  
 }  
  
 **int**[][] res = findArrayOfShortestWaysBellman(newVertex, numberOfNodes + 1, numberOfNodes + 1);  
 **if** (res.length == 0)  
 **return new int**[0][0];  
 **int**[] h = res[0];  
  
 **for** (**int** i = 0; i < vertex.length; i++) {  
 vertex[i][2] += h[vertex[i][0] - 1] - h[vertex[i][1] - 1];  
 }  
  
  
 **int**[][] newAdjacencyMatrix = getAdjacencyMatrix();  
  
  
 res = findArrayOfShortestWaysDijkstra(newAdjacencyMatrix, node);  
 **int**[] distances = res[0];  
 **int**[] arrayP = res[1];  
 **int**[] newDistances = **new int**[distances.length];  
  
 **for** (**int** i = 0; i < newDistances.length; i++) {  
 newDistances[i] = distances[i] + h[i] - h[node];  
 }  
 **return new int**[][]{newDistances, arrayP};  
  
 }  
  
  
 **public void** findShortestWaysBellman(**int** node) {  
 **int**[][] res = findArrayOfShortestWaysBellman(vertex, numberOfNodes, node);  
 **if** (res.length == 0)  
 **return**;  
 **int**[] distances = res[0];  
 **for** (**int** i = 0; i < distances.length; i++) {  
 **if** (i != node-1)  
 System.*out*.println((node) + "->" + (i + 1) + ": " + ((distances[i] == MAX\_VALUE) ? "∞" : distances[i]));  
 }  
 }  
  
 **public void** findShortestWayBellman(**int** node1, **int** node2) {  
 **int**[][] res = findArrayOfShortestWaysBellman(vertex, numberOfNodes, node1);  
 **if** (res.length == 0)  
 **return**;  
 **int**[] distances = res[0];  
 **int**[] arrayP = res[1];  
 **if**(distances[node2-1] == MAX\_VALUE){  
 System.*out*.println("There\'s no way between these nodes!");  
 **return**;  
 }  
 System.*out*.println("Minimal distance: " + distances[node2 - 1]);  
 System.*out*.println("Way: " + Arrays.*toString*(findWay(arrayP, node1, node2)));  
 }  
  
  
 **private int**[][] findArrayOfShortestWaysBellman(**int**[][] vertex, **int** numberOfNodes, **int** node) {  
 node--;  
 **int**[] arrayP = **new int**[numberOfNodes];  
 Arrays.*fill*(arrayP, -1);  
 **int**[] distanceArray = **new int**[numberOfNodes];  
 **for** (**int** i = 0; i < distanceArray.length; i++) {  
 **if** (i != node)  
 distanceArray[i] = MAX\_VALUE;  
 }  
  
 **for** (**int** k = 0; k < numberOfNodes - 1; k++) {  
// System.out.println(Arrays.toString(distanceArray));  
  
 **for** (**int** i = 0; i < vertex.length; i++) {  
 **if** (distanceArray[vertex[i][0] - 1] != MAX\_VALUE &&  
 distanceArray[vertex[i][0] - 1] + vertex[i][2] < distanceArray[vertex[i][1] - 1]) {  
 distanceArray[vertex[i][1] - 1] = distanceArray[vertex[i][0] - 1] + vertex[i][2];  
 arrayP[vertex[i][1] - 1] = vertex[i][0];  
 }  
 }  
   
 }  
  
 **for** (**int** i = 0; i < vertex.length; i++) {  
 **if** (distanceArray[vertex[i][0] - 1] != MAX\_VALUE &&  
 distanceArray[vertex[i][0] - 1] + vertex[i][2] < distanceArray[vertex[i][1] - 1]) {  
 System.*out*.println("Graph has minus cycles!");  
 **return new int**[0][0];  
 }  
 }  
  
 **return new int**[][]{distanceArray, arrayP};  
  
  
 }  
  
 **int**[] findWay(**int**[] arrayP, **int** node1, **int** node2) {  
 ArrayList<Integer> list = **new** ArrayList<>();  
 **while** (node2 != -1) {  
 list.add(node2);  
 node2 = arrayP[node2 - 1];  
 }  
 **return** getInvertedArrayFromList(list);  
 }  
  
 **private int**[] getInvertedArrayFromList(ArrayList<Integer> list) {  
 **int**[] arr = **new int**[list.size()];  
// System.out.println(list.toString());  
 **for** (**int** i = list.size() - 1; i >= 0; i--) {  
 arr[list.size() - i - 1] = list.get(i);  
 }  
// System.out.println(Arrays.toString(arr));  
 **return** arr;  
 }  
  
 **public void** findShortestWayDijkstra(**int** node1, **int** node2) {  
 node1--;  
 node2--;  
 **int**[][] res = findArrayOfShortestWaysDijkstra(**this**.adjacencyMatrix, node1);  
 **int**[] distances = res[0];  
 **int**[] arrayP = res[1];  
 **if** (distances[node2] == MAX\_VALUE) {  
 System.*out*.println("There\'s no way between these nodes!");  
 **return**;  
 }  
 System.*out*.println(Arrays.*toString*(arrayP));  
 **int**[] way = findWay(arrayP, node1 + 1, node2 + 1);  
// for (int i = 0; i < way.length; i++)  
// way[i]++;  
 System.*out*.println("Minimal distance: " + distances[node2]);  
 System.*out*.println("Way: " + Arrays.*toString*(way));  
 }  
  
 **public void** findShortestWaysDijkstra(**int** node) {  
 node--;  
 **int**[] distances = findArrayOfShortestWaysDijkstra(**this**.adjacencyMatrix, node)[0];  
 System.*out*.println("Minimal distances: ");  
 **for** (**int** i = 0; i < distances.length; i++) {  
 **if** (i != node)  
 System.*out*.println((node + 1) + "->" + (i + 1) + ": " + ((distances[i] == MAX\_VALUE) ? "∞" : distances[i]));  
 }  
 }  
  
 **private int**[] findWayDijkstra(**int**[] distances, **int** node1, **int** node2) {  
// System.out.println("KU" + Arrays.toString(distances));  
 **int** startDistance = distances[node2];  
 ArrayList<Integer> nodesOnWay = **new** ArrayList<>();  
 **while** (startDistance != 0) {  
 **for** (**int** i = 0; i < numberOfNodes; i++) {  
 **if** (i != node2 && **this**.adjacencyMatrix[i][node2] != MAX\_VALUE &&  
 startDistance - **this**.adjacencyMatrix[i][node2] == distances[i]) {  
 nodesOnWay.add(node2);  
 node2 = i;  
 startDistance = distances[node2];  
 **break**;  
 }  
 }  
 }  
 nodesOnWay.add(node1);  
 **return** getInvertedArrayFromList(nodesOnWay);  
 }  
  
 **private boolean** hasFalses(**boolean**[] arr) {  
 **for** (**int** i = 0; i < arr.length; i++)  
 **if** (!arr[i])  
 **return true**;  
  
 **return false**;  
 }  
  
  
 **private int**[][] findArrayOfShortestWaysDijkstra(**int**[][] adjacencyMatrix, **int** node) {  
  
 **int**[] arrayP = **new int**[numberOfNodes];  
 Arrays.*fill*(arrayP, -1);  
 **for** (**int** i = 0; i < numberOfNodes; i++) {  
 **if** (i != node && adjacencyMatrix[node][i] != MAX\_VALUE) {  
 arrayP[i] = node + 1;  
 }  
 }  
  
 **boolean**[] doneNodes = **new boolean**[adjacencyMatrix.length];  
 Arrays.*fill*(doneNodes, **false**);  
  
 **int**[] distanceArray = **new int**[numberOfNodes];  
 Arrays.*fill*(distanceArray, MAX\_VALUE);  
  
 distanceArray[node] = 0;  
 **while** (hasFalses(doneNodes)) {  
 doneNodes[node] = **true**;  
  
// System.out.println(Arrays.toString(distanceArray));  
// System.out.println(Arrays.toString(arrayP));  
// System.out.println(Arrays.toString(distanceArray) + " " + node + " " + Arrays.toString(doneNodes));  
 **for** (**int** i = 0; i < distanceArray.length; i++) {  
 **if** (i != node && adjacencyMatrix[node][i] != MAX\_VALUE) {  
 **if** (distanceArray[i] > (distanceArray[node] + adjacencyMatrix[node][i])) {  
 distanceArray[i] = distanceArray[node] + adjacencyMatrix[node][i];  
 arrayP[i] = node + 1;  
 }  
  
 }  
 }  
  
 **int** closestNode = -1;  
 **for** (**int** i = 0; i < distanceArray.length; i++) {  
 **if** (i != node && !doneNodes[i]) {  
 **if** (closestNode == -1 || distanceArray[i] < distanceArray[closestNode])  
 closestNode = i;  
 }  
 }  
  
 **if** (closestNode == -1)  
 **break**;  
  
 node = closestNode;  
// System.out.println(Arrays.toString(distanceArray));  
  
 }  
 **return new int**[][]{distanceArray, arrayP};  
 }  
  
 **private int**[][] getAdjacencyMatrix() {  
 **int**[][] adjacencyMatrix = **new int**[**this**.numberOfNodes][**this**.numberOfNodes];  
 **for** (**int** i = 0; i < adjacencyMatrix.length; i++) {  
 Arrays.*fill*(adjacencyMatrix[i], MAX\_VALUE);  
 }  
 **for** (**int**[] ints : **this**.vertex) {  
 adjacencyMatrix[ints[0] - 1][ints[1] - 1] = ints[2];  
 }  
 **for** (**int** i = 0; i < numberOfNodes; i++)  
 adjacencyMatrix[i][i] = 0;  
 **return** adjacencyMatrix;  
 }  
  
  
 }  
  
 **class** MyStack {  
 **int**[] mStack;  
 **int** lastIndex = -1;  
  
 MyStack(**int** length) {  
 mStack = **new int**[length];  
 //mStack[0] = first;  
 }  
  
 **int** getLength() {  
 **return** mStack.length;  
 }  
  
 **int** getCurrentNode() {  
 **if** (lastIndex >= 0)  
 **return** mStack[lastIndex];  
 **else return** -1;  
 }  
  
  
 **void** put(**int** node) {  
 lastIndex++;  
 mStack[lastIndex] = node;  
 }  
  
 **boolean** removeLast() {  
 **if** (lastIndex != -1) {  
 mStack[lastIndex] = 0;  
 lastIndex--;  
 **return false**;  
 }  
 **return true**;  
  
 }  
  
 String getString() {  
 StringBuilder output = **new** StringBuilder();  
 **for** (**int** i = 0; i <= lastIndex; i++) {  
 output.append(mStack[i] + 1).append(" ");  
 }  
 **return** output.toString();  
 }  
 }

Результати роботи програми:



