Лабораторна робота №3

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

**Завдання:**

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

на вхід програми подається вхідний файл з описом неорієнтованого графу, зі структурою,

яка вказана у практичному завданні No1 «Представлення графів». При реалізації

алгоритмів вважати, що заданий граф є зв’язаним.

1. Обійти граф пошуком вшир. Користувач вводить початкову вершину графу.

Програма виконує обхід графу, починаючи з вказаної початкової вершини. На

екран виводиться протокол обходу – таблиця, яка містить наступні дані по кожній

ітерації алгоритму обходу: поточна вершина, її BFS-номер, вміст черги (див. тему

28 електронного конспекту).

2. Обійти граф пошуком углиб. Аналогічно за пунктом 1 завдання, але програма

виконує обхід графу пошуком углиб. На екран виводиться протокол обходу:

поточна вершина, її DFS-номер, вміст стеку.

Код:

**package** com.company;  
  
**import** java.io.File;  
**import** java.io.FileNotFoundException;  
**import** java.util.Scanner;  
  
**public class** DS\_IP92\_LR3\_MedvedievM {  
  
 **public static void** main(String[] args) **throws** FileNotFoundException {  
 UndirectedGraph graph = **new** UndirectedGraph(**new** File("inputs/neorient.txt"));  
 System.out.print("BFS or DFS?: ");  
 Scanner scanner = **new** Scanner(System.in);  
 String choice = scanner.nextLine();  
 System.out.print("Enter start node index: ");  
 **int** startIndex = scanner.nextInt();  
  
 **if** (choice.equals("BFS"))  
 graph.printBFS(startIndex);  
 **else if** (choice.equals("DFS")) {  
 graph.printDFS(startIndex);  
 }  
 }  
}  
  
**abstract class** Graph {  
 **protected int**[][] verges;  
 **protected int** numberOfNodes, numberOfVerges;// n вершин, m ребер  
 **protected int**[][] adjacencyMatrix;  
  
 **protected** Graph(File file) **throws** FileNotFoundException {  
 parseFile(file);  
 preSetAdjacencyMatrix();  
 }  
  
 **private void** parseFile(File file) **throws** FileNotFoundException {  
 Scanner fileScanner = **new** Scanner(file);  
 **this**.numberOfNodes = fileScanner.nextInt();  
 **this**.numberOfVerges = fileScanner.nextInt();  
 **this**.verges = **new int**[**this**.numberOfVerges][2];  
 **for** (**int** i = 0; i < **this**.numberOfVerges; i++) {  
 verges[i][0] = fileScanner.nextInt();  
 verges[i][1] = fileScanner.nextInt();  
 }  
 }  
  
  
 **protected void** preSetAdjacencyMatrix() {  
 **this**.adjacencyMatrix = **new int**[**this**.numberOfNodes][**this**.numberOfNodes];  
 }  
  
}  
  
**class** UndirectedGraph **extends** Graph {  
  
 **protected** UndirectedGraph(File file) **throws** FileNotFoundException {  
 **super**(file);  
  
 }  
  
 @Override  
 **protected void** preSetAdjacencyMatrix() {  
 **super**.preSetAdjacencyMatrix();  
 **for** (**int** i = 0; i < **this**.numberOfVerges; i++) {  
 **this**.adjacencyMatrix[**this**.verges[i][0] - 1][**this**.verges[i][1] - 1] = 1;  
 **this**.adjacencyMatrix[**this**.verges[i][1] - 1][**this**.verges[i][0] - 1] = 1;  
 }  
 }  
  
 **private int** recursNumber;  
 **boolean**[] doneNodes;  
  
 **public void** printDFS(**int** startIndex) {  
 MyStack stack = **new** MyStack(numberOfNodes);  
 startIndex--;  
 recursNumber = 1;  
 doneNodes = **new boolean**[stack.getLength()];  
 stack.put(startIndex);  
 doneNodes[startIndex] = **true**;  
 System.out.println("Node: " + (startIndex + 1) + ", DFC-number: " + recursNumber + ", stack: " + stack.getString());  
 dfsRecurs(stack);  
 }  
  
 **private void** dfsRecurs(MyStack stack) {  
 **int** currentNode = stack.getCurrentNode();  
 **if** (currentNode >= 0) {  
 **for** (**int** i = 0; i < numberOfNodes; i++) {  
 **if** (currentNode != i && adjacencyMatrix[currentNode][i] == 1 && !doneNodes[i]) {  
 stack.put(i);  
 recursNumber++;  
 doneNodes[i] = **true**;  
 System.out.println("Node: " + (i + 1) + ", DFS-number: " + recursNumber + ", stack: " + stack.getString());  
 dfsRecurs(stack);  
 }  
 }  
  
 **boolean** isEmpty = stack.removeLast();  
 **if** (!isEmpty) {  
 System.out.println("Node: " + "-" + ", DFS-number: " + "-" + ", stack: " + stack.getString());  
 dfsRecurs(stack);  
 }  
 }  
 }  
  
  
 **public void** printBFS(**int** startIndex) {  
 MyQueue queue = **new** MyQueue(numberOfNodes);  
 startIndex--;  
 recursNumber = 1;  
 queue.put(startIndex);  
 doneNodes = **new boolean**[queue.getLength()];  
 doneNodes[startIndex] = **true**;  
 System.out.println("Node: " + (startIndex + 1) + ", BFS-number: " + recursNumber + ", queue: " + queue.getString());  
 bfsResurs(queue);  
 }  
  
 **private void** bfsResurs(MyQueue queue) {  
 **int** currentNode = queue.getCurrentNode();  
 **if** (currentNode != -1) {  
 **for** (**int** i = 0; i < numberOfNodes; i++) {  
 **if** (currentNode != i && adjacencyMatrix[currentNode][i] == 1 && !doneNodes[i]) {  
 queue.put(i);  
 recursNumber++;  
 doneNodes[i] = **true**;  
 System.out.println("Node: " + (i + 1) + ", BFS-number: " + recursNumber + ", queue: " + queue.getString());  
 }  
 }  
 queue.removeFirst();  
 System.out.println("Node: " + "-" + ", BFS-number: " + "-" + ", queue: " + queue.getString());  
 bfsResurs(queue);  
 }  
 }  
  
  
}  
  
**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();  
 }  
}  
  
**class** MyQueue {  
 **int**[] mQueue;  
 **int** lastIndex = -1, firstIndex = 0;  
  
 MyQueue(**int** length) {  
 mQueue = **new int**[length];  
 }  
  
 **int** getLength() {  
 **return** mQueue.length;  
 }  
  
 **void** put(**int** node) {  
 lastIndex++;  
 mQueue[lastIndex] = node;  
 }  
  
 **void** removeFirst() {  
 firstIndex++;  
 }  
  
 String getString() {  
 StringBuilder output = **new** StringBuilder();  
 **for** (**int** i = firstIndex; i <= lastIndex; i++) {  
 output.append(mQueue[i] + 1).append(" ");  
 }  
 **return** output.toString();  
 }  
  
 **int** getCurrentNode() {  
 **if** (firstIndex < mQueue.length)  
 **return** mQueue[firstIndex];  
 **else return** -1;  
 }  
  
  
}

Результати роботи програми:

