Project 2

Graph components and testing

Made by student
2IiE-SP
Volodymyr Chuchka

1. Introduction

The project includes the implementation of the** Graph **class representing an undirected graph and an additional** GraphException **class for handling graph-related exceptions. Additionally, the** GraphTester **class was created for testing the project's functionality. Below, we discuss the implementation and present test results for the** components() **method on three sample graphs, each with at least 10 vertices.

2. Graph Class

The** Graph **class represents an undirected graph and contains functions for manipulating the graph. Key elements of this class include:

–`addVertex()`: Adding a new vertex.

```
public int addVertex() {
    adjLists.add(new ArrayList<>());
    return adjLists.size() - 1;
}
```

–`order()`: Returning the number of vertices in the graph.

```
public int order() {
    return adjLists.size();
}
```

–`degree(int v)`: Returning the degree of a vertex.

```
public int degree(int v) {
    if (v >= order())
        throw new IllegalArgumentException("No such vertex");
    return adjLists.get(v).size();
}
```

- `getNeighboursOf(int v)`: ***Returning neighbors of a vertex.***

```
public ArrayList<Integer> getNeighboursOf(int v) {
    if (v >= order())
        throw new IllegalArgumentException("No such vertex");
    return new ArrayList<Integer>(adjLists.get(v));
}
```

- `isEdge(int v, int w)`: ***Checking if an edge exists.***

```
public boolean isEdge(int v, int w) {
    if (v >= order() || w >= order())
        throw new IllegalArgumentException("No such vertex");
    if (this.degree(w) < this.degree(v))
        for (int x : this.getNeighboursOf(w)) {
            if (x == v)
                return true;
        }
    else
        for (int x : this.getNeighboursOf(v)) {
            if (x == w)
                return true;
        }

    return false;
```

- `addEdge(int v, int w)`: ***Adding an edge.***

```
public void addEdge(int v, int w) {
    if (this.isEdge(v, w))
        throw new IllegalArgumentException("Such edge already exists!");
    else {
        adjLists.get(v).add(w);
        adjLists.get(w).add(v);
    }
}
```

- `print()`: ***Displaying information about the graph.***

```
public void print() {
    System.out.println(adjLists);
}
```

- `isConnected()`: **Checking if the graph is connected.**

```
public boolean isConnected() {
    if (this.order() == 0)
        return false;
    int s = 0;

    boolean[] visited = new boolean[this.order()];
    ArrayDeque<Integer> queue = new ArrayDeque<>();
    queue.addLast(s);
    visited[s] = true;
    int numberOfVisited = 1;
    while (!queue.isEmpty()) {
        int v = queue.poll();
        for (int w : this.getNeighboursOf(v))
            if (!visited[w]) {
                visited[w] = true;
                numberOfVisited++;
                queue.addLast(w);
            }
    }
    if(numberOfVisited==this.order()) return true;
    else return false;

}
```

- `isConnectedV2()`: **Alternative method for checking connectivity.**

```
public boolean isConnectedV2() {
    if (this.order() == 0)
        return false;
    int s = 0;
    HashSet<Integer> visited=new HashSet<>();
    ArrayDeque<Integer> queue = new ArrayDeque<>();
    queue.addLast(s);
    visited.add(s);
    while (!queue.isEmpty()) {
        int v = queue.poll();
        for (int w : this.getNeighboursOf(v))
            if (!visited.contains(w)) {
                visited.add(w);
                queue.addLast(w);
            }
    }
    if(visited.size()==this.order()) return true;
    else return false;
}
```

-`components()`: Finding connected components.

```
private void DFScontinue(int v, boolean[] visited, List<Integer> component) {
    visited[v] = true;
    component.add(v);
    for (int neighbor : adjLists.get(v)) {
        if (!visited[neighbor]) {
            DFScontinue(neighbor, visited, component);
        }
    }
}
 public List<List<Integer>> components() {
        boolean[]visited = new boolean[order()];
        List<List<Integer>> components = new ArrayList<>();

        for (int ver = 0; ver < adjLists.size(); ver++) {
            if (!visited[ver]) {
                List<Integer> component = new ArrayList<>();
                DFScontinue(ver, visited, component);
                components.add(component);
            }
        }
        return components;
    }
```

3. GraphException Class

The GraphException class extends the Exception class and is used to handle exceptions related to errors in the project. It was implemented for situations where it's not possible to create a graph based on input data.

4. Tester Class

The GraphTester class serves as a tester for the project. Its main function is to generate three random graphs(generateRandomGraph()) with at least 10 vertices each and test the components() method on each of them. Test results are displayed in the console.

In test class i have one helping class which called Pair and i am implementing some utils for generating random graphs

```
static class Pair<T, U> {
    public final T first;
    public final U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }
}
```

```
import java.util.HashSet;
import java.util.Random;
import java.util.Set;
```

—-*generateRandomGraph() : Creating random graph*

```
private static Graph generateRandomGraph(int vertices) {
    Graph randomGraph = new Graph();
    for (int i = 0; i < vertices; i++) {
        randomGraph.addVertex();
    }
    Random random = new Random();
    Set<Pair> addedEdges = new HashSet<>();

    for (int i = 0; i < vertices * 2; i++) {
        int v = random.nextInt(vertices);
        int w = random.nextInt(vertices);
        while (v == w || addedEdges.contains(new Pair(v, w)) || addedEdges.contains(new Pair(w, v)) || randomGraph.isEdge(v, w)) {
            v = random.nextInt(vertices);
            w = random.nextInt(vertices);
        }
        addedEdges.add(new Pair(v, w));
        randomGraph.addEdge(v, w);
    }

    return randomGraph;
}
```

5. Implementation of the components()Method

The components() method in the Graph class uses the Depth-First Search (DFS) algorithm to identify connected components in the graph. During the iteration through the graph vertices, for each unvisited vertex, the DFS function is called, which recursively visits all neighbors of the vertex, creating a component list. Each found component is added to the list of all components in the graph.

6. Results of Testing

```
public static void main(String[] args) {
     Graph graph1 = generateRandomGraph(10);
        Graph graph2 = generateRandomGraph(10);
        Graph graph3 = generateRandomGraph(10);

        // Test and print connected components for each graph
        System.out.println("Connected Components for Graph 1:");
        System.out.println(graph1.components());

        System.out.println("Connected Components for Graph 2:");
        System.out.println(graph2.components());

        System.out.println("Connected Components for Graph 3:");
        System.out.println(graph3.components());
    }
```

Results

```
Connected Components for Graph 1:
[[0, 1, 6, 2, 4, 5, 3, 8, 9, 7]]
Connected Components for Graph 2:
[[0, 7, 5, 9, 2, 1, 3, 8, 6, 4]]
Connected Components for Graph 3:
[[0, 2, 1, 9, 5, 7, 4, 3, 8, 6]]
```