GIT Department of Computer Engineering
CSE 222/505 - Spring 2017
Homework 09
Due date: 19.05.2017 – 23:55

SCENARIO:

In this homework, you will create a new abstract class named AbstractGraphExtended by extending AbstractGraph abstract class of the book. The new abstract class will provide new features by implementing the methods described below.

public int addRandomEdgesToGraph (int *edgeLimit*);

This method selects a random number between 0 and *edgeLimit* then adds that much edges to calling graph. The source and destination vertices of edges will be random again. The inserted edges will be directed or undirected according to calling graph (If graph is directed, new edges will be directed vice versa). The weights of edges will be set to 1. It will return number of inserted edges.

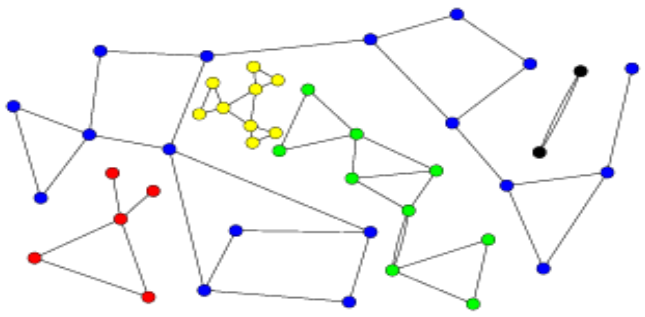
public int [] breadthFirstSearch (int *start*);

This method performs a breadth first search on calling graph starting from vertex *start* and returns an array which contains the predecessor of each vertex in the breadth-first search tree. It will work for both directed and undirected graphs.

public Graph [] getConnectedComponentUndirectedGraph ();

This method finds *connected components* in a graph, creates ListGraph or MatrixGraph instances for each connected component and returns them in a Graph array. If this method is called from a ListGraph instance, then connected components will be ListGraph instances. Same thing will happen for MatrixGraph instance. The method will only work for undirected graphs.

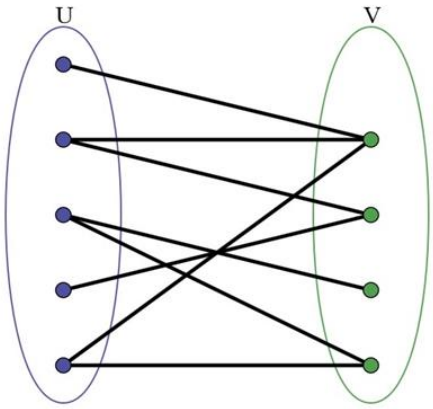
A *connected component* is a set of vertices that is reachable from anyone to all others in the set. Your goal is to find all connected components in a graph, put each of them to new graph instances with their edges then return it. Here is an example, suppose the input graph is the one below:



This is just a single graph with 5 different connected components. So, you will find and return 5 graphs. Hint: If you are tired of homeworks, eat some bread for energy.

public boolean isBipartiteUndirectedGraph ();

This method returns true if calling graph instance is *bipartite* graph, false otherwise. A *bipartite graph* is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. Here is an example of bipartite graph:



Like previous method, this one will also work for only undirected graphs. Hint: Maybe eating bread again will be good.

public void writeGraphToFile (String fileName);

This method will do the reverse of createGraph method in the AbstractGraph abstract class and write number of vertices, source and destination vertex of each edge in a file. An example file output will be like this:

```
9
0 1
0 3
1 2
1 4
1 5
2 5
3 6
4 6
4 7
5 7
6 8
7 8
```

First line indicates number of vertices in graph, other lines represent edges of the graph. For example, there is an edge from 0 to 1, 0 to 3 and 1 to 2 etc. Don’t write weights of edges.

You will test each of your methods using both ListGraph and MatrixGraph classes of book by creating instances of them in main method. These classes will extend AbstractGraphExtended abstract class which you implemented instead of AbstractGraph abstract class of book.

Test each method two times by reading two different graphs from two different files. You can use createGraph method of book to read graphs from file. Write the result graphs to files by using the writeGraphToFile method (do not test writeGraphToFile method exclusively) for testing first and third methods. Output the result of second and fourth methods to console. You will follow this test procedure for both ListGraph and MatrixGraph. Don't use big graphs, at most 10 vertices will be enough.

You will send us an IntelliJ IDEA project that includes

- Main class and AbstractGraphExtended abstract class that you implemented
- Input graph files for your tests
- Edge class, Graph interface, AbstractGraph abstract class, ListGraph and MatrixGraph classes. They are all added to zip file.

And of course your report where you explained the implemented new methods.

RESTRICTIONS:

- Can be only one main class in project
- Only using classes and libraries from Graphs chapter of your book is allowed. Don't use any other third part library.

GENERAL RULES:

- For any question firstly use course news forum in moodle, and then the contact TA.
- You can submit assignment one day late and will be evaluated over twenty percent (%20).
- Register github student pack and create private project and upload your projects into github.
- Your appeals are considered over your github project process.

TECHNICAL RULES:

- Use given CSE222-VM to develop and test your homeworks (your code must be working on CSE222-VM), CSE222-VM download link will be given on Moodle.
- Implement clean code standarts in your code;
    - Classes, methods and variables names must be meaningful and related with the functionality.
    - Your functions and classes must be simple, general, reusable and focus on one topic.
    - Use standart java code name conventions.

REPORT RULES:

- Add all javadoc documentations for classes, methods, variables …etc. All explanation must be meaningful and understandable.
- You should submit your homework code, javadoc and report to Moodle in a studentid_hw#.tar.gz file.
- Use the given homework format including selected parts:

Detailed system requirements	

The Project usecase diagrams (extra points)	
Class diagrams	
Other diagrams	
Problem solutions approach	x
Test cases	x
Running command and results	x

GRADING :

- No error handling : -50
- No inheritance : -95
- No javadoc documentation : -50
- No report : -90
- Disobey restrictions : -100
- **Cheating : -200**
- Your solution is evaluated over 100 as your performance.

CONTACT :

Ahmet SOYYİĞİT