Data Structures and Algorithms
Spring 2020
Assignment 3

IT University Innopolis

Contents

1 About this homework

1.1 Coding part

For practical (coding) part you have to provide you program as a single Java class file that contains all necessary definitions and a `main` method.

The program should read from standard input and write to standard output unless mentioned otherwise.

1.1.1 Preliminary tests

For some coding parts a CodeForces or CodeTest system will be used for preliminary tests. You will have to specify a submission number in those systems as a proof that your solution is correct. Only correct solutions will be assessed further by TAs.

1.1.2 Code style and documentation

The source code should contain adequate internal documentation in the form of comments. Internal documentation should explain how the code has been tested, e.g. the various scenarios addressed in the test cases.

Do not forget to include your name in the internal documentation.

All important exceptions should be handled properly.

Bonus points might be awarded for elegant solutions, great documentation and the coverage of test cases. However, these bonus points will only be able to cancel the effects of penalties.

1.2 Theoretical part

Solutions to theoretical exercises have to be elaborate and clear. Write all solutions in a single document and make it a PDF for submitting.

Do not forget to include your name in the document.

1.3 Submission

You are required to submit your solutions to all parts via Moodle.

For this assignment you will need to submit:

- Java class file(s) for the coding parts;
- submission number or a link to submission in CodeForces as a proof of correctness;
- a PDF file with solutions to theoretical parts.

Submit files as a single archive with your name and group number. E.g. `BS18-00_Ivanov_Ivan.zip`.

1.4 Plagiarism

Plagiarism will not be tolerated and a plagiarised solutions will be heavily penalised for all parties involved. Remember that you learn nothing when you copy someone else's work which defeats the purpose of the exercise!

You are allowed to collaborate on general ideas with other students as well as consult books and Internet resources. However, be sure to credit all the sources you use to make it clear what part of your solution comes from elsewhere.

2 Coding part (70 point)

In this part of the assignment you are required to implement a representation for undirected graphs. In two problems you will be given unweighted graphs as input and in the third problem input will be a weighted graph.

Graph representation should be implemented as a generic Java class.

Coding style for the entire coding part is graded out of 10 points.

2.1 Connected graph (15 points)

Implement a method `analyzeConnectivity()` that will check if a graph is connected, meaning that for every two vertices u and v in a graph there exists a path from u to v. This method should somehow indicate if the graph is connected or not, and in case it is not connected it should provide a counterexample — at least two vertices of the graph that are not connected by a path.

Document method by summarizing the idea behind your implementation and by adding a time complexity annotation in terms of both number of vertices and edges.

Using this method, write a program that analyzes connectivity of a given graph. Input graph is given in standard input as follows:

- first line contains two numbers N (number of vertices) and M (number of edges) separated by space symbol;
- then each of the next M lines contains two numbers i and j — indices of vertices (vertices are indexed from 1 to N), connected by an edge.

Input graph does not have loops or parallel edges.

If input graph is connected, the program should print `GRAPH IS CONNECTED` to standard output. Otherwise, the program should output

`VERTICES <U> AND <V> ARE NOT CONNECTED BY A PATH`

where `<U>` and `<V>` are replaced with vertice indices from the input graph.

Input	Output
3 2 1 2 2 3	GRAPH IS CONNECTED
4 2 1 2 4 3	VERTICES 1 AND 3 ARE NOT CONNECTED BY A PATH

2.2 Components dictionary (15 points)

Implement a method `vertexComponents()` that returns a dictionary that maps every vertex of a graph into a connected component it belongs to. You can choose any suitable data structure to use for dictionary.

Document method by summarizing the idea behind your implementation and by adding a time complexity annotation in terms of both number of vertices and edges.

Using this method implement a program that reads a graph and a list of vertex indices from standard input and outputs an index of corresponding component of a graph for each of input vertices.

Input is given as follows:

- first line contains two numbers N (number of vertices) and M (number of edges) separated by space symbol;
- then each of the next M lines contains two numbers i and j — indices of vertices (vertices are indexed from 1 to N), connected by an edge.

Input graph does not have loops or parallel edges.

Output should contain N numbers, ith number should indicate an index of a component for vertex i.

Input	Output
3 2 1 2 2 3	1 1 1
4 2 1 2 4 3	2 2 1 1

2.3 Minimum spanning forest (30 point)

Implement a method `minimumSpanningForest()` that returns a minimum spanning forest of a *weighted* graph. You choose any suitable data structure to represent the forest. You should use either Prim's algorithm or Kruskal's algorithm.

Document method by summarizing the idea behind your implementation and by adding a time complexity annotation in terms of both number of vertices and edges.

Using this method implement a program that reads a weighted graph from standard input and outputs some minimum spanning forest.

Input is given as follows:

- first line contains two numbers N (number of vertices) and M (number of edges) separated by space symbol;
- then each of the next M lines contains three numbers
  - i — index of "from" vertex ();
  - j — index of "to" vertex;
  - w — weight of an edge ($1 < w < 10^5$).

Input graph does not have loops or parallel edges.

First line of output should contain a single number F — number of trees in a forest. The following lines describe trees in the forest. For ith tree

- the first line contains numbers T_i (number of vertices in ith tree) and R_i (index of any vertex of that tree);

- each of the next T_i lines describe an edge of ith tree: a triple of numbers $from_i$, to_i and $weight_i$.

Input	Output
3 3 1 2 11 2 3 13 1 3 12	1 3 2 1 2 11 1 3 12
4 2 1 2 10 4 3 81	2 2 1 1 2 10 2 4 4 3 81

3 Theoretical part (30 points)

3.1 Constructing B-tree (10 points)

Show the resultant B-tree (minimum degree $t = 2$) after inserting the following keys in order:

$$9, 31, 42, 69, 2, 26, 34, 57, 76, 27, 58$$

3.2 The square graph (20 points)

The square of a directed graph $\mathcal{G} = (V, E)$ is the graph $\mathcal{G}_2 = (V, E_2)$ such that $(u, w) \in E_2$ iff there exists $v \in V$ such that $(u, v) \in E$ and $(v, w) \in E$; i.e., there is a path of exactly two edges from u to w. Give algorithms and their respective time complexities using the Big-O notation for constructing the square graph. Your solution should consist of algorithm 1 (using adjacency list) and its complexity and algorithm 2 (using adjacency matrix) and its complexity.