Data Structures and Algorithms
Spring 2021
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 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 code documentation!

All important exceptions should be handled properly.

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

1.2 Theoretical part

Solutions to theoretical exercises have to be elaborate and clear. Write all solutions in a single document, then save or compile it as 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;
- link(s) to accepted submission(s) in CodeForces as a proof of correctness;
- a PDF file with solutions for theoretical parts.

Submit files as a single archive with your name and group number. For example, if your name is *Ivanov Ivan* and your group is *BS20-05* then your archive should be named `BS20-05_Ivanov_Ivan.zip`.

2 Coding part (70 points)

2.1 Building graphs (30 points)

Define graph ADT and its implementation based on adjacency matrix. In particular, you need to implement the following:

1. `Graph` ADT (as a Java interface or an abstract class in C++), supporting the following methods:
    1. `addVertex(value)` — add a vertex with value `value` to the graph and return reference to the created vertex object;
    2. `removeVertex(v)` — remove a vertex, provided a reference to a vertex object;
    3. `addEdge(from, to, weight)` — add a (directed) edge between `from` and `to` vertices with weight `weight`, return reference to the created edge object;
    4. `removeEdge(e)` — remove an edge, given a reference to an edge object;
    5. `edgesFrom(v)` — return a collection or edge objects that are going from vertex `v`;
    6. `edgesTo(v)` — return a collection or edge objects that are going into vertex `v`;
    7. `findVertex(value)` — find any vertex with the specified value;
    8. `findEdge(from_value, to_value)` — find any edge with specified values in the source and target vertices.
    9. `hasEdge(v, u)` — determine whether there exists a directed edge from `v` to `u`;
2. `Vertex` class of vertex objects;
3. `Edge` class of edge objects;
4. `AdjacencyMatrixGraph` class implementing `Graph` ADT using adjacency matrix;
5. All of the above should be generic in the type of values stored at vertices and weights of edges.

After implementing all of the above, write a program that inputs instructions from the standard input to create, modify and query a graph. Here is a list

of instructions:

1. `ADD_VERTEX <name>` — add a vertex with a given name[1];
2. `REMOVE_VERTEX <name>` — remove a vertex with a given name;
3. `ADD_EDGE <from_name> <to_name> <weight>` — add an edge from `from_name` to `to_name` with an integer weight `weight`;
4. `REMOVE_EDGE <from_name> <to_name>` — remove an edge from `from_name` to `to_name`;
5. `HAS_EDGE <from_name> <to_name>` — output `TRUE` if there is an edge from `from_name` to `to_name` and `FALSE` otherwise;

Sample input:

```
ADD_VERTEX A
ADD_VERTEX B
ADD_EDGE A B 3
HAS_EDGE A B
HAS_EDGE B A
REMOVE_EDGE A B
HAS_EDGE A B
```

Sample output:

```
TRUE
FALSE
FALSE
```

2.2 Cycle detection and transposition (20 points)

Add the following methods to your Graph implementation from the previous section:

1. `isAcylcic()` — determine whether graph is acyclic;
2. `transpose()` — reverse all edges in a graph.

Extend the program from the previous section to support two more instructions:

1. `IS_ACYCLIC` — output `ACYCLIC` if the graph is acylcic, otherwise output `<weight> <v1> <v2> ... <vN>` where `<weight>` is the total weight of

[1] a name may consist of alphanumeric symbols (such as `Example1` or `test123`).

the cycle and `<v1> <v2> ... <vN>` is a sequence of vertices constituting a cycle;

2. `TRANSPOSE` — transpose the graph.

Sample input:

```
ADD_VERTEX A
ADD_VERTEX B
ADD_VERTEX C
ADD_EDGE A B 3
ADD_EDGE B C 4
TRANSPOSE
IS_ACYCLIC
ADD_EDGE A C 5
IS_ACYCLIC
```

Sample output:

```
ACYCLIC
12 A C B
```

2.3 Shortest paths (20 points)

You are given a graph of a computer network where each edge has a length in meters and bandwidth. Implement a program using `AdjacencyMatrixGraph` to compute the shortest path from host A to host B with a requirement of minimum bandwidth W. You need to use Dijkstra's algorithm to solve this.

First line of the input contains two numbers `N M`, where N is the number of vertices and M is the number of edges. The following M lines specify edges with four integer numbers `i_k j_k l_k b_k` where i_k and j_k are indices of source and target vertices, l_k is length of kth edge and b_k is bandwidth of the edge. Last line contains two integer indices, for start and finish vertices.

Sample input:

```
3 3
1 2 1 1
1 3 2 2
3 2 2 3
1 3
```

If it is impossible to find the solution to a given problem, the program should output `IMPOSSIBLE`. Otherwise, first line of the output should contain three numbers `n l b`, where n is the number of vertices in the path, l — total length of the path and b — bandwidth of the path. Second line should contain n indices of vertices constituting the path.

Sample output:

```
3 4 2
1 3 2
```

3 Theoretical part (30 points)

3.1 Minimum spanning tree (20 points)

Consider a special undirected weighted graph $G_n(V, E)$:

1. G_n contains n vertices ($n > 2$).
2. Vertices v_i and v_j ($i \neq j$) are connected by an edge if and only if
   1. either $|i - j| \leq 2$ and $\left\lfloor \frac{i-1}{4} \right\rfloor = \left\lfloor \frac{j-1}{4} \right\rfloor$
   2. or $|i - j| \leq 4$ and $i + j \equiv 0 \mod 4$
3. Each edge (v_i, v_j) is assigned a weight[2] of $i + j + (|i - j| - 1)^2$.

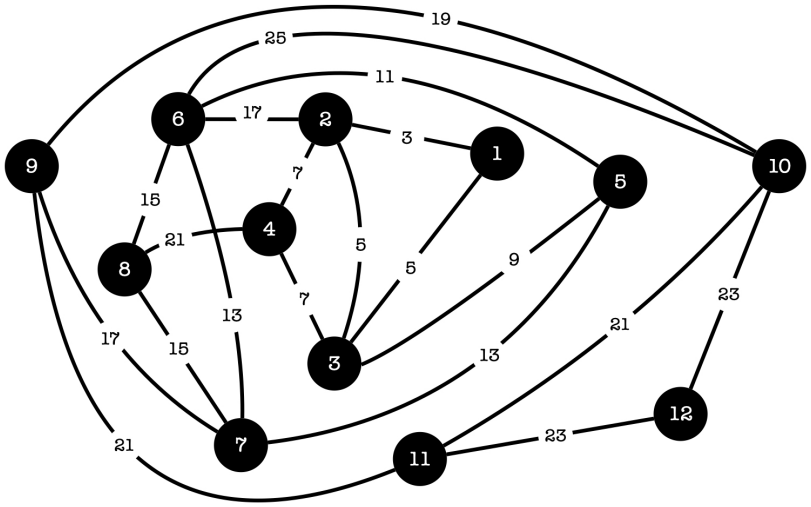

Figure 1: Graph G_{12} with $n = 12$.

Do the following:

1. Write down the total cost of minimum spanning tree (MST) for graph G_n in general. Justify your answer.
2. Verify your answer by building MST for the sample graph G_{12}, using Kruskal's OR Prim's Algorithm:

[2] if $i = 5$ and $j = 7$ then $i + j + (|i - j| - 1)^2 = 12 + (2 - 1)^2 = 13$.

1. Specify which algorithm you are using.
2. Show all the steps and provide illustration of intermediate MST at each of the first 3 steps[3] and last 3 steps of the algorithm.
3. Provide a clear explanation of the state of the algorithm (show the state/content of any variables or data structures used) after step 6 (i.e. after 6 edges have been added) and explain which edge should be added next and why.

3.2 Shortest path (10 points)

Suggest an adjustment to Floyd-Warshall algorithm that will maintain shortest paths under addition of vertices and edges:

1. Explain how to store the state (information about shortest paths).
2. Write pseudocode for updating the state:
    1. after adding a vertex;
    2. after adding an edge;
3. Compute the time complexity of the pseudocode in the previous point. Justify your answer.

[3] one step of the algorithm adds an edge to the intermediate tree or forest.