TD7- Introduction to Graphs

Part 0: the Java environment for graphs

To complete this lab and the upcoming labs on graphs, you must add all the classes provided in Moodle to your environment. Take some time to thoroughly review the graph classes provided and understand the purpose of the available methods. Pay particular attention to the following items:

- the *Vertex* interface: this interface describes the available methods on vertices
- the *Edge* interface: this interface describes the available methods on edges
- the *Edge* interface: this interface describes the available methods on edges
- the *Graph* interface: this interface describes all the available methods on graphs (both directed and undirected graphs). Some of these methods have the suitable behavior depending on the type of graph they are applied to (directed or undirected graphs)
- the *GraphReader* class: this is a convenient class to easily build a graph from a String representation.

The other classes may not to be fully understood to do the labs because all the available public methods are gathered in the three interfaces mentioned before. However, it could be interesting to look to:

- the *DiGraph* class: this class is used to build directed graphs.
- the *UnDiGraph* class: this class is used to build undirected graphs.

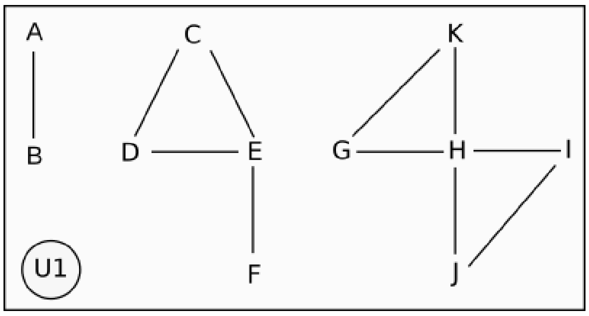
Some Test[1] classes are given to ensure the given code is running in your environnement : *GraphReaderTest* which define lab examples, and *TestGraph*.

[1] Pas vraiment, certains tests sont juste des affichages, n’hésitez pas à les améliorer.

Finally, to do this lab you are to use the class skeletons: *ConnectedComponents* and *GraphHelper* and some tests to check your codes : *ConnectedComponentsTest and GraphHelperTest*

Part 1: connected components

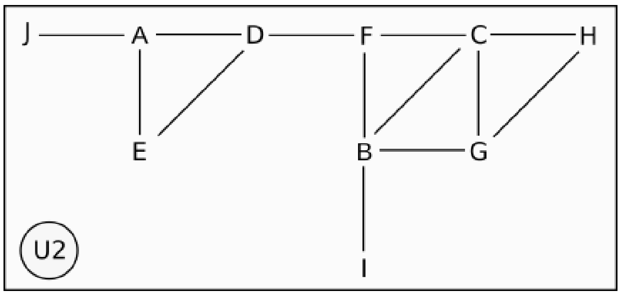
In this part, you are to write the method *find* from the class *ConnectedComponent* to compute the connected components of an **undirected** graph. The connected components are the subsets of connected vertices. A graph of **n** vertices may have between **1** (connected graph) and **n** (graph with no edge) connected components. Your program should return a dictionary (an object from class Map<Vertex,Integer>) *CC* such that *CC.get(v)* is the number of the connected component the vertex *v* belongs to. The actual value of *CC.get(v)* is not relevant as long as *CC.get(u)* = *CC.get(v)* if and only if *u* and *v* are in the same connected component. For example, for the graph U1:



the method *find* should return one dictionary such as (3 groups of connected components): {H=1, B=2, D=3, J=1, K=1, I=1, F=3, A=2, E=3, G=1, C=3}

(En résumé, A et B appartiennent au même groupe de composants et le numéro associé à ce groupe est 2 (le nombre n'a pas d'importance), les nœuds D, E,C, F appartiennent au groupe 3, etc.)

and for the graph U2:



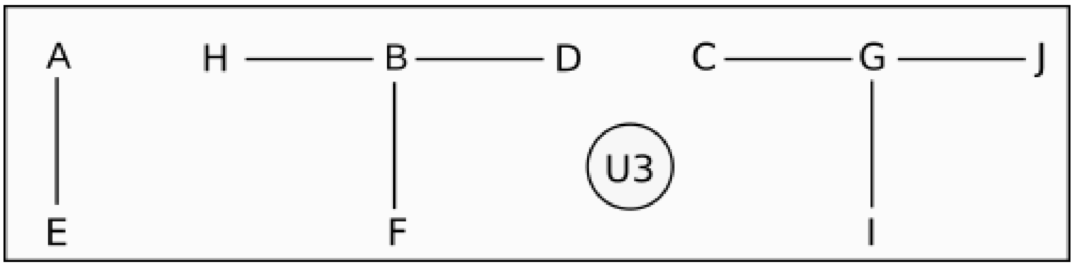
this method should return the following dictionary:

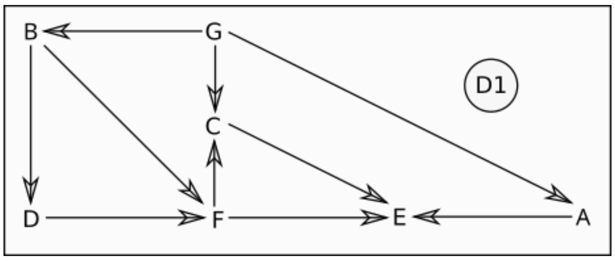
{C=1, B=1, D=1, J=1, H=1, F=1, A=1, G=1, E=1, I=1}

Notice that the dictionary produced by your function may not display the vertices with the same ordering and the number **of** the connected components may be different. The complexity of the method *find* must be O(|V| + |E|) where V is the set of all vertices and E the set of all edges.

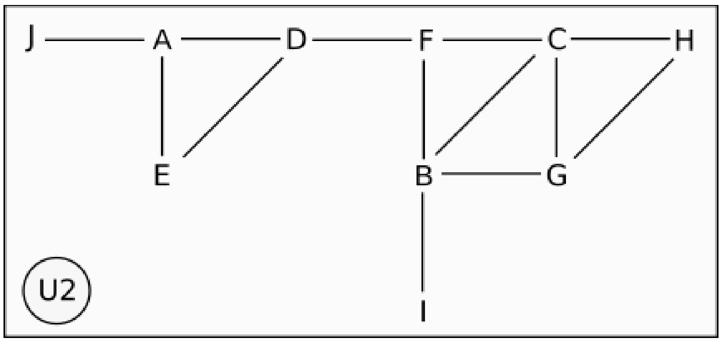
Part 2: finding cycles

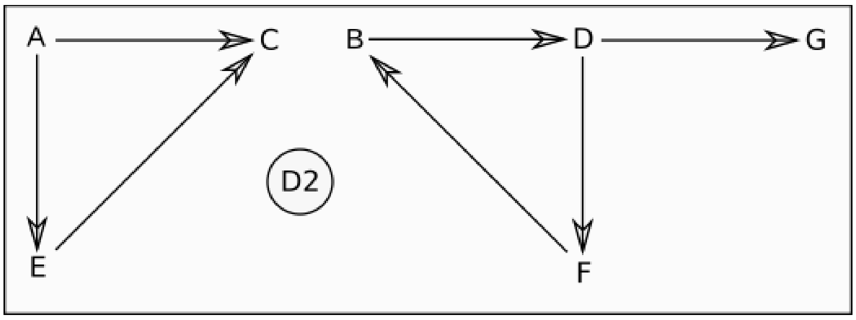
In this part, you will write the methods *hasCycle* from class *GraphHelper* to check if a graph has a cycle. You must write two pairs of methods, one for **undirected** graphs and one for **directed** graphs. The public methods *hasCycle* return a *boolean*: the returned value is *true* if the graph has a cycle, and *false* otherwise (i.e., if the graph is **acyclic**). For examples, the following graphs are **acyclic**:





and the following graphs are **cyclic**:



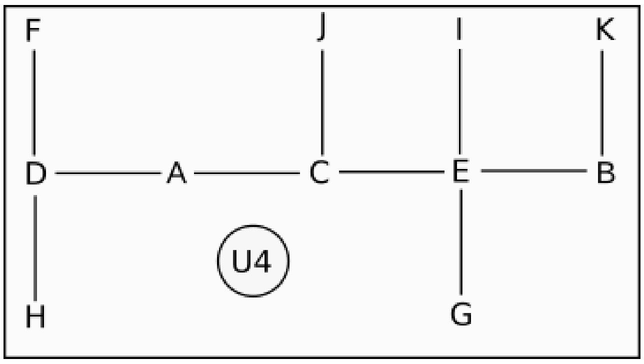


The complexity of the public method *hasCycle* must be O(|V| + |E|) where V is the set of all vertices and E the set of all edges.

Part 3: finding path

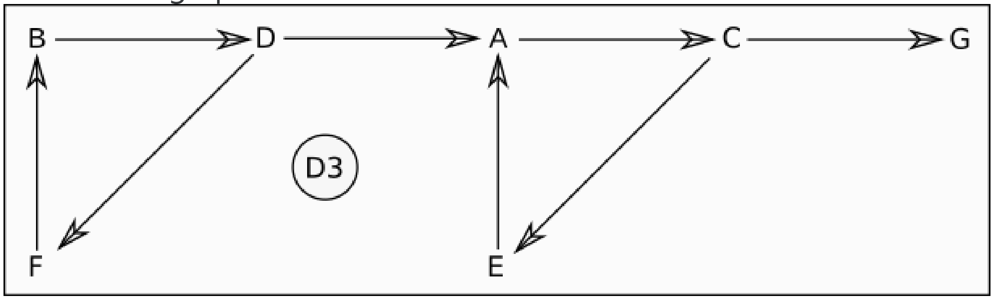
In this part, you are to write the method *findPath* to find a *path* in a graph between two vertices. Your method should work for **both** undirected and directed graphs. In case there is at least a path from vertex u to vertex v in the graph, the function will return one of the possible paths from u to v as a list [u_1, u_2, ..., u_n] where u_1=u and u_n=v. If there is no path from u to v in the graph, the function path returns the empty list [].

For example, given U4 the following graph :



findPath(U4,U4.getVertex("H"),U4.getVertex("K")) returns [H, D, A, C, E, B, K]

and for the graph D3:



findPath(D3,D3.getVertex("F"),D3.getVertex("E")) returns [[F, B, D, A, C, E]

although *findPath(D3,D3.getVertex("E"),D3.getVertex("B"))* returns [ ].

The complexity of the method *findPath* must be O(|V| + |E|) where V is the set of all vertices and E the set of all edges.

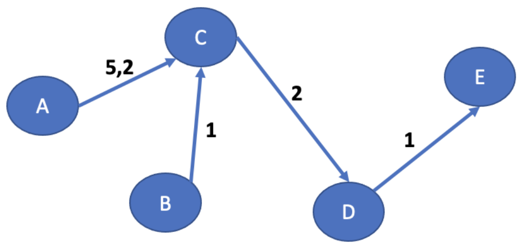
The path from A to A is [A].

Part 4: Computing the distance of a path

You will calculate the distance of a path. By default, we have assigned a weight of 1 to each edge (see in the code), so for graphs where the edges do not have weights, the distance will be the number of edges traversed. Otherwise, it will be the sum of the distances.

For example, in D3, distance([H, D, A, C, E, B, K]) = 6

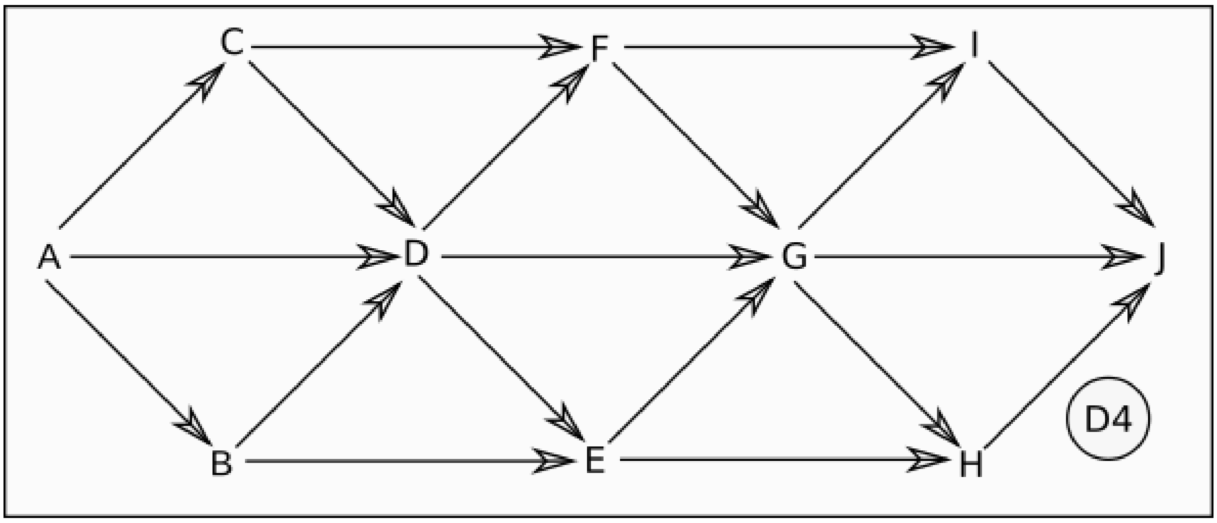
According to the following definition, distance([A,C,D,E]) = 8.2



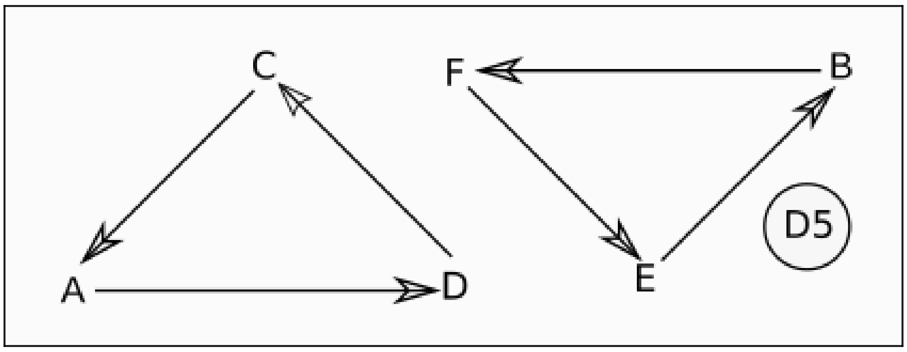
You may have already noticed that the distance between two nodes varies depending on the path taken.

Part 5: finding root

Given a **directed** graph, a *root* is a vertex from which there is at least one path to all other vertices. For example, the graph D1 from part 2 has one root (the vertex G), the graph D3 form part 3 has 3 roots (B, D and F) but graph D2 from part 2 has no root. Another example is the graph D4 which has one root (vertex A).



The graph D5 doesn't have a root.



Write the method *findRoot* from class *RootFinder* to check if a **directed** graph has root. Your method should return a root if the graph has at least one root, *null* otherwise. A major constraint about your method is that its complexity should be **again O(|V| + |E|)** ! The algorithm you have to design works as follow :

- first find a candidate: a candidate is a vertex such that *It is not reachable from any other vertex*
- check is the candidate is a root: for a vertex u to be a root, there must be a path from u to all other vertices in the graph

- both previous steps must be in $O(|V| + |E|)$