Algorithms & Data Structures
SI3 - Polytech Nice-Sophia - Edition 2018

# Lab #7: Introduction to graphs

This assignment will give you practice about basic graph algorithms. For all parts you are to use the provided classes:

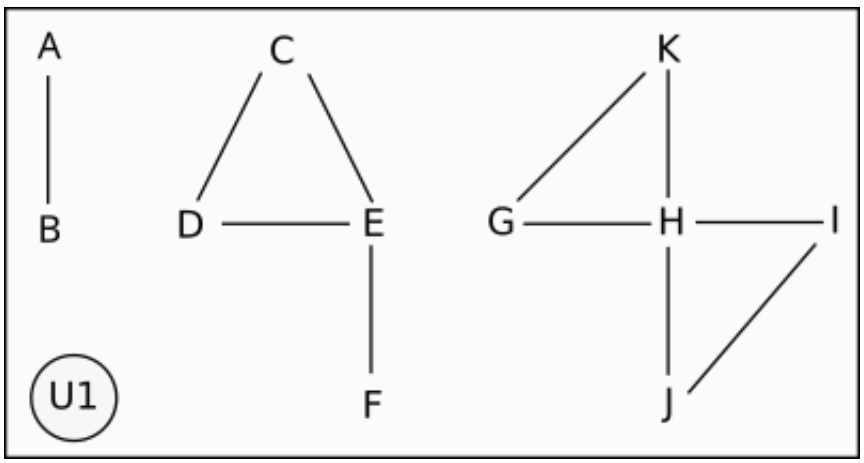
- lab7.py: a testing module. This file provides function skeletons you have to complete plus some interactive code you can use to test them
- graph.py: a module for the `Graph` class and utility functions on graphs

## Part 0: the Graph class

In this part you should just take a few minutes to review the provided class `Graph` and try to understand the role of the methods available.

## Part 1: connected components

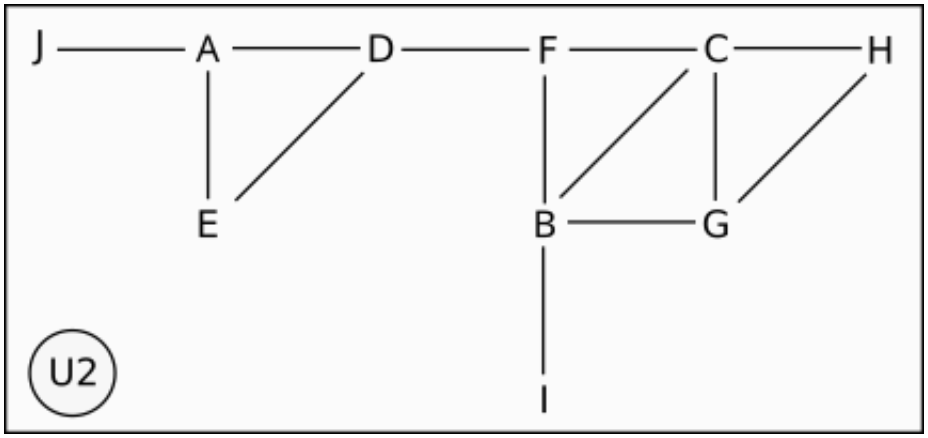
In this part, you are to write the function `connected_components` to compute the connected components of an **undirected** graph. The connected components are the subsets of connected vertices. A graph of `n` vertices may have between `1` (connected graph) and `n` (graph with no edge) connected components. Your program should return a dictionary `CC` such that `CC[v]` is the number of the connected component the vertex `v` belongs to. The actual value of `CC[v]` is not relevant as long as `CC[u] = CC[v]` if and only if `u` and `v` are in the same connected component. For example, for the graph `U1`




the function `connected_components` should return the following dictionary:

```
{K: 1, J: 1, B: 2, H: 1, I: 1, G: 1, E: 3, C: 3, A: 2, D: 3, F: 3}
```

and for the graph `U2`:



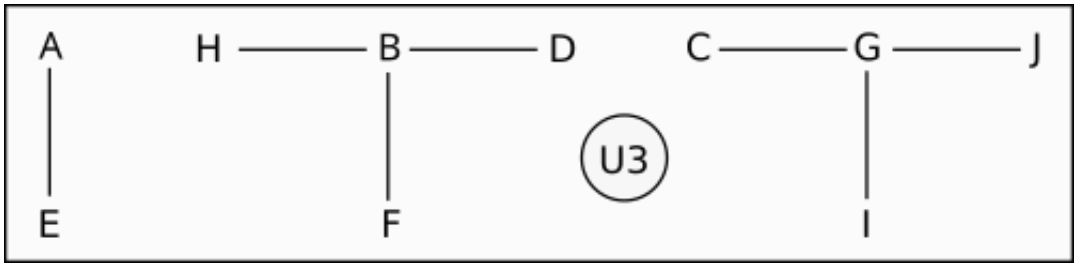

the function `connected_components` should return the following dictionary:

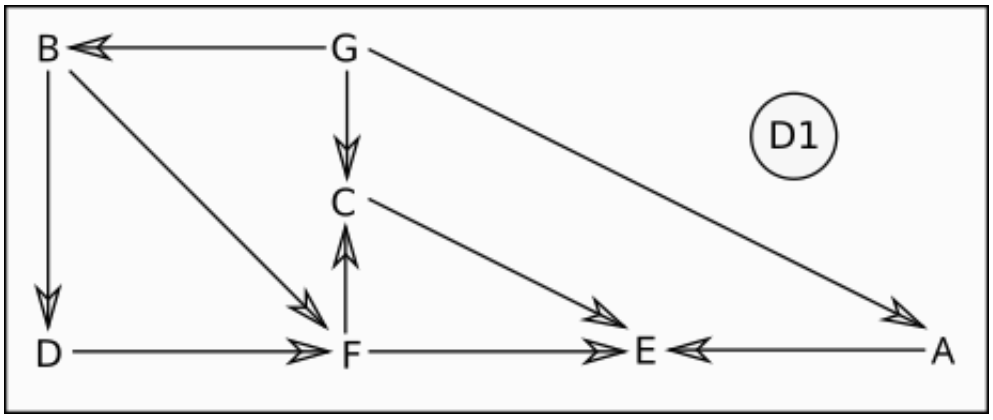
```
{I: 1, B: 1, C: 1, A: 1, F: 1, D: 1, H: 1, J: 1, G: 1, E: 1}
```

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 function `connected_components` must be O(|V| + |E|) where V is the set of all vertices and E the set of all edges.

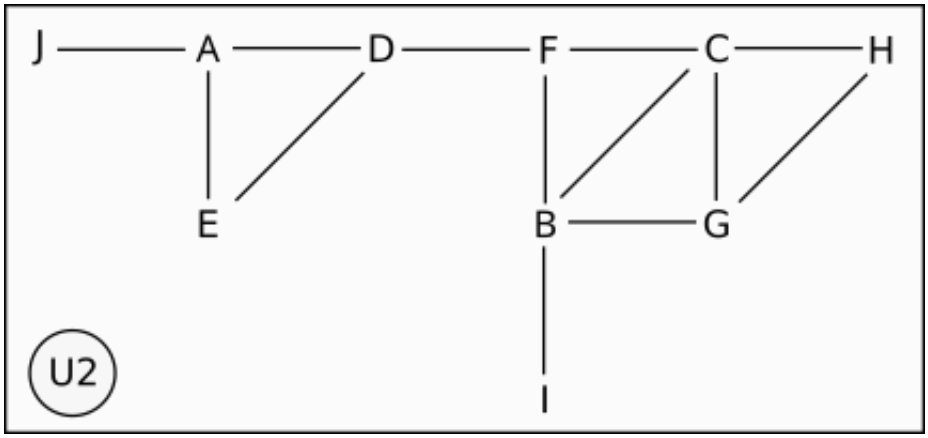
# Part 2: finding cycles

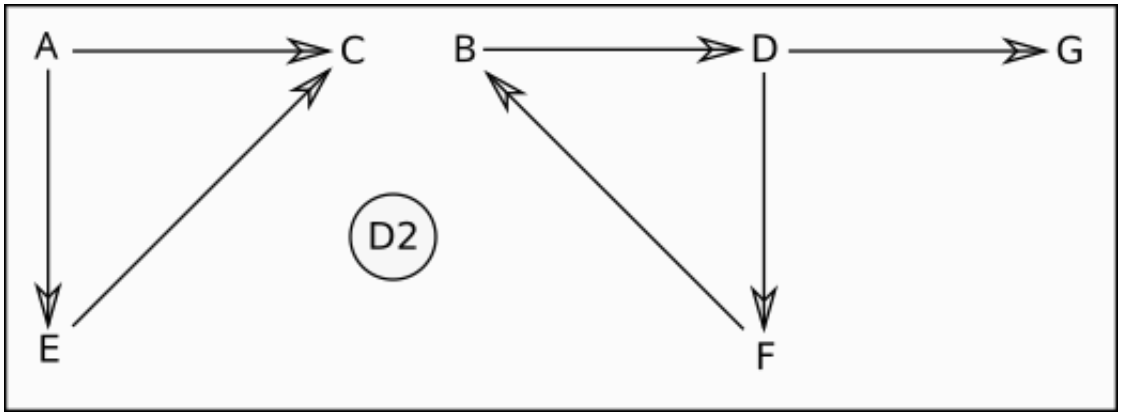
In this part, you are to write the function `has_cycle` to check if a graph has a cycle. You must write two functions, one for **undirected** graphs and one for **directed** graphs. Each function returns a *boolean*: the returned value is `True` if the graph has a cycle, `False` otherwise (i.e. if the graph is **acyclic**). For example, The following graphs are **acyclic**:




and the following graphs are **cyclic**:

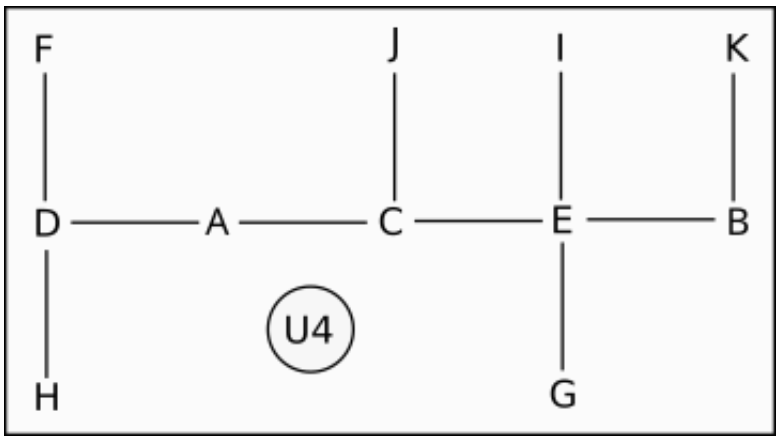



The complexity of the function `has_cycle` must be O(|V| + |E|) where V is the set of all vertices and E the set of all edges.

# Part 3: finding path

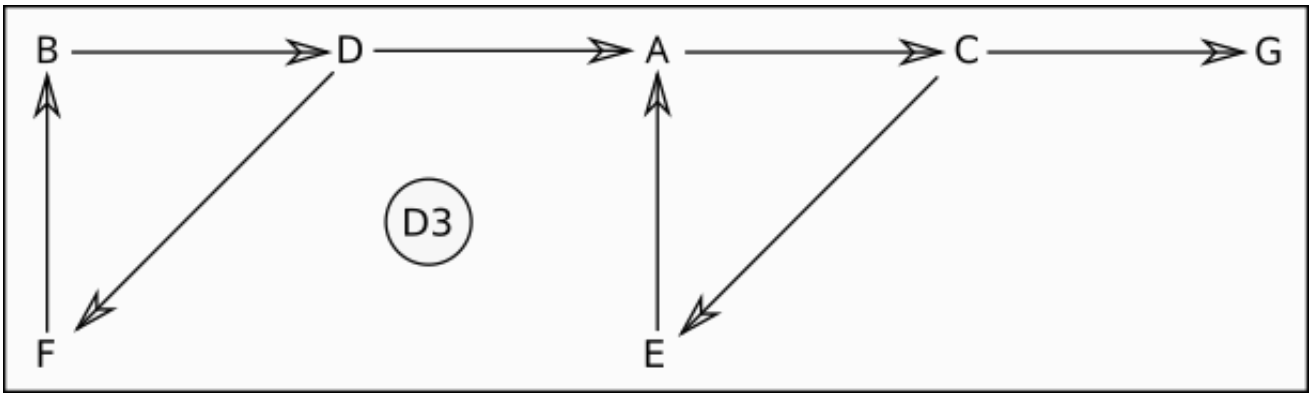
In this part, you are to write the function `path` to find a *path* in a graph between two vertices. Your function should work for **both** undirected and directed graphs. In case there is at least a path from vertex `u` to vertex `v` in the graph, the function will return one of the possible path from `u` to `v` as a list `[ u1, u2, ..., un ]` where `u1 = u` and `un = 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:




`path(U4,U4.get_vertex('H'),U4.get_vertex('K'))` returns:

```
[H, D, A, C, E, B, K]
```

and for the the graph `D3`:




`path(D3,D3.get_vertex('F'),D3.get_vertex('E'))` returns:

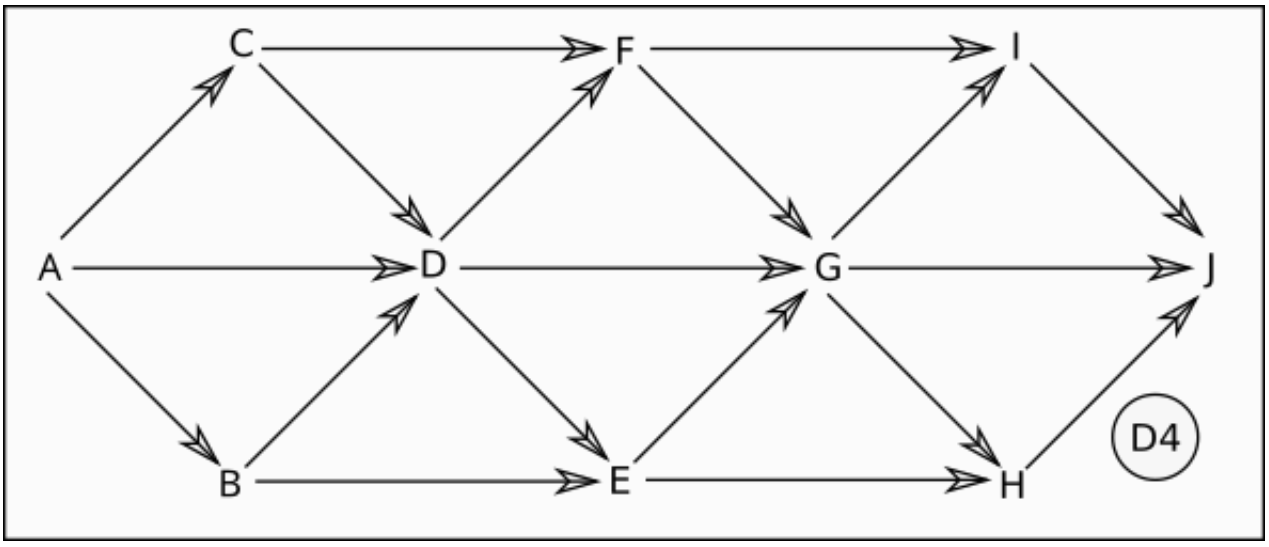
```
[F, B, D, A, C, E]
```

although `path(D3,D3.get_vertex('E'),D3.get_vertex('B'))` returns `[]`.

The complexity of the function `path` must be O(|V| + |E|) where V is the set of all vertices and E the set of all edges.

# Part 4: 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`).

Write the function `has_root` to check if a **directed** graph has root. Your function should return a root if the graph has at least one root, `None` otherwise. A major constraint about your function `has_root` 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, if the graph has a root, the candidate is a root, otherwise the candidate is just a random vertex of the graph
- 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|)