Algorithmics
Midterm #4 (C4)

Undergraduate 2nd year (S4)
Epita

1 March 2017 - 9 : 30

Instructions (read it) :

☐ You must answer on **the answer sheets provided.**

- No other sheet will be picked up. Keep your rough drafts.
- Answer within the provided space. **Answers outside will not be marked**: Use your drafts!
- Do not separate the sheets unless they can be re-stapled before handing in.
- Penciled answers will not be marked.

☐ The presentation is negatively marked, which means that you are marked out of 20 points and the presentation points (maximum of 2) are taken off this grade.

☐ **Code:**

- All code must be written in the language `Python` (no C, Caml, Algo or anything else).
- **Any `Python` code not indented will not be marked.**
- All that you need (class, types, routines) is indicated in the appendix (last page). **Read it!**
- You can write your own functions as long as they are documented (we have to know what they do).
  In any case, the last written function should be the one which answers the question.

☐ Duration : 2h

Exercise 1 (SCC – *4 points*)

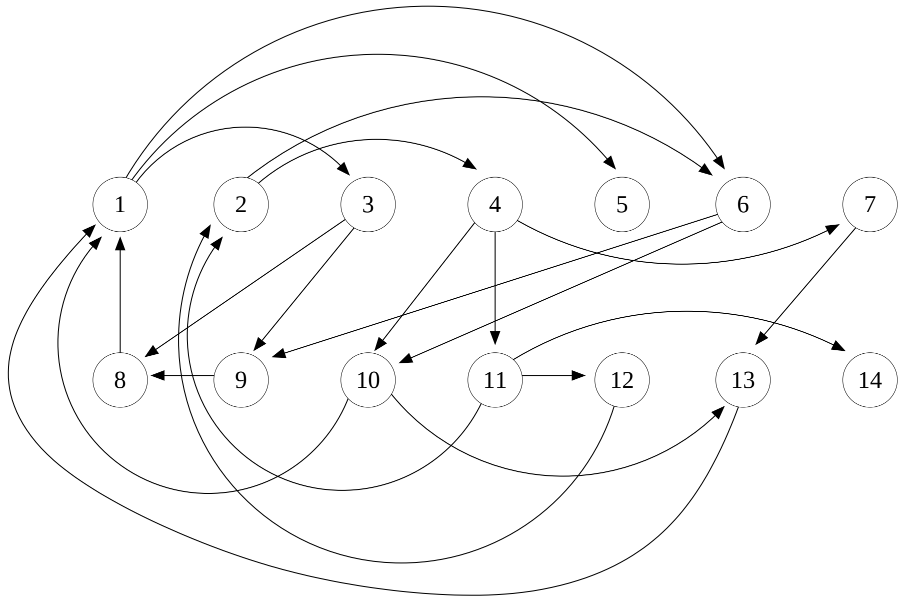


Figure 1: Directed graph.

1. Draw the spanning forest associated to the depth-first traversal of the graph in figure 1. The vertex 1 will be considered as the first of the traversal and the successors will be treated in ascending order. We will draw only the discovery edges.
2. How many strongly connected components does this graph contain?
3. Indicate the different strongly connected components of that graph. To facilitate the illustration, you will just color the vertices of the same component with the same color (one color per component). No need to show arcs.

Exercise 2 (Biconnectivity – *2 points*)

In this exercise we work with simple undirected graphs without a loop.

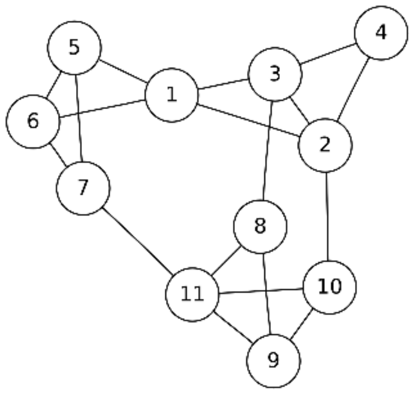


Figure 2: Example of graph

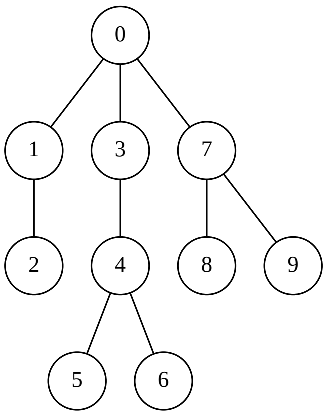
1. Recall the definition of biconnectivity.
2. Is the graph in figure 2 biconnected?
3. Give the definition of an articulation point.
4. Give the definition of an isthmus.

Exercise 3 (Even Tree – *5 points*)

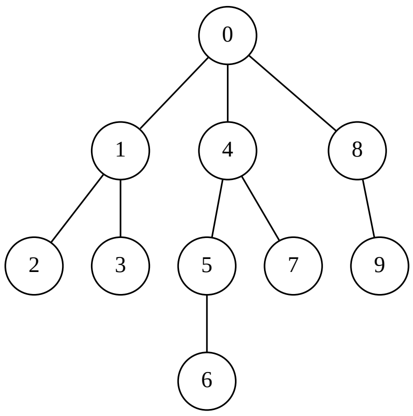
You are given a *tree* (a simple connected graph with no cycles). The aim here is to find the **maximum** number of edges you can remove from the tree to get a forest such that each connected component of the forest contains an **even** number of vertices.

Note: The given graph is such that it can always be decomposed into components containing an even number of vertices.

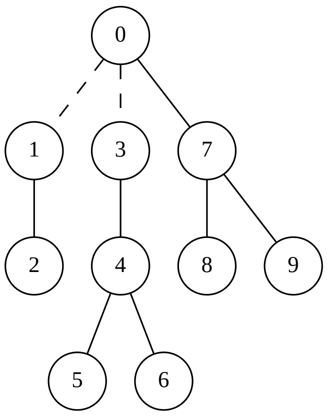
For instance, during a depth-first traversal, we found:

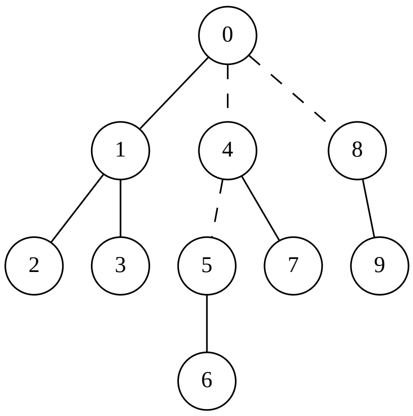


2 edges can be removed:
{0, 1} and {0, 3}



3 edges can be removed:
{4, 5}, {0, 4} and {0, 8}





In both cases, the result is a forest with connected components of even size.

Note: it works whatever the starting vertex is.

Write the function `evenTree(`G`)` that computes the maximum number of edges that can be removed from G according to the previous specifications.

In the next exercises, we call *connected component vector* of the graph $G = < S, A >$ containing k connected components, the vector cc such that $\forall s \in S, cc[s]$ is the number of the connected component to which s belongs (an integer between 1 and k).

Exercise 4 (Warshall – ***3,5 points*****)**

Reminder:

- We call *transitive closure* of a graph G defined by the couple $< S, A >$, the graph $G*$ defined by the couple $< S, A* >$ such that for all pairs of vertices $x, y \in S$, there exists an edge $\{x, y\}$ in $G*$ if-and-only-if there exists a chain from x to y in G.
- Warshall algorithm computes the adjacency matrix of the transitive closure of a graph.

Given M, the result matrix of Warshall applied to the graph G, write the function `CCFromWarshall(M)` that returns the pair (k, cc): k is the number of connected components of G, and cc its connected component vector (a list in Python).

Exercise 5 (Union-Find – ***3 points*****)**

Let G be the graph $< S, A >$, n its number of vertices and L its list of edges (a list of vertex pairs). Write the function `CCFromEdges(n, L)` that returns the pair (k, cc): k is the number of connected components of G, and cc its connected component vector (a list in Python).

Exercise 6 (Journey To The Moon – ***3,5 points*****)**

The member states of the UN are planning to send 2 people to the Moon. In line with their principles of global unity, they want to pair astronauts of different countries.

There are N trained astronauts numbered from 0 to $N - 1$. But those in charge of the mission did not receive information about the citizenship of each astronaut. The only information they have is that some particular pairs of astronauts belong to the same country.

Your task is to compute in how many ways they can pick a pair of astronauts belonging to different countries. Assume that you are provided enough pairs to let you identify the groups of astronauts even though you might not know their country directly. For instance, if 1, 2, 3 are astronauts from the same country; it is sufficient to mention that (1, 2) and (2, 3) are pairs of astronauts from the same country without providing information about a third pair (1, 3).

L is a list of pairs (A, B) such that A and B, both in $[0, N-1]$, are astronauts from the same country. Examples :

- With $N = 5$, $L =$ [(0, 1), (0, 2), (3, 4)], the result is $6 = 3 * 2$
- With $N = 9$, $L =$ [(0, 1), (0, 2), (1, 3), (4, 5), (6, 7), (7, 8)], the result is $26 = 4 * 2 + 4 * 3 + 2 * 3$

Write the function `Moon(N, L)` that computes in how many different pairs of astronauts can be picked.

Appendix

Graphs

The graph in exercise 4 is represented with adjacency lists.

```python
class Graph:
    def __init__(self, order, directed = False):
        self.order = order
        self.directed = directed
        self.adjLists = []
        for i in range(order):
            self.adjLists.append([])
```

Functions you can use

- on lists: `len`, `append`
- `range`

Given functions

```python
def find(x, p):
    rx = x
    while p[rx] >= 0:
        rx = p[rx]
    return rx

def union(x, y, p):
    rx = find(x, p)
    ry = find(y, p)
    if rx != ry:
        if p[rx] < p[ry]:
            p[rx] = p[rx] + p[ry]
            p[ry] = rx
        else:
            p[ry] = p[ry] + p[rx]
            p[rx] = ry

def build(L, n):
    '''
    n: integer > 0
    L: list of pairs (a, b) with a and b in [0, n[
    '''
    p = [-1]*n
    for (x, y) in L:
        union(x, y, p)
    return p
```

Your functions

You can write your own functions as long as they are documented (we have to know what they do).
In any case, the last written function should be the one which answers the question.