Exercise Session n. 10

Algorithms and Data Structures

We practice a bit with graphs. All algorithms are based on the adjacency-list representation of a graph. Vertices are numbers from 0 to $n-1$. Thus `G[u]` is an array of numbers for every vertex `u`. Notice that an undirected graph represents an edge (u, v) with two directed edges, such that `G[u]` contains `v` and `G[v]` contains `u`.

For the purpose of testing, you may use the following function to read from a file object.

```
def read_graph(f):
    """Read a graph from a file object 'f' (text) containing one
    vertex and its adjacency list per line.  E.g.:

    Input:     | Graph:
    A B C      |  A --> B
    B C        |  |    /^
    C B        |  v   / |
               |  C<-/  /
               |   ^---/

    Return three containers:

    Name: (array) Vertex Id -> Vertex Name
    Adj: (array) Vertex Id -> array of Vertex Id
    Idx: (dictionary) Vertex Name -> Vertex Id
    """
    Name = []
    Adj = []
    Idx = {}
    for line in f:                  # for each line in the input f
        l = line.strip().split()
        assert len(l) > 0
        u_name = l[0]               # v_name is the source vertex
        if u_name in Idx:           # We already have vertex v_name
            u = Idx[v_name]
        else:
            u = len(Name)           # Add vertex at the end of Name,
            Idx[u_name] = u
            Name.append(u_name)
            Adj.append([])
```

```
        for i in range (1, len(l)):
            v_name = l[i]        # u_name is a target vertex
            if v_name in Idx:    # We already have vertex v_name
                v = Idx[v_name]
            else:                # Add vertex, as above
                v = len(Name)
                Idx[v_name] = v
                Name.append(v_name)
                Adj.append([])
            Adj[u].append(v)

    return Name, Adj, Idx
```

Counting Connected Components

Write an algorithm `count_connected_components(G)` that takes an *undirected* graph G and returns the number of connected components in G.

Topological Sort

Write an algorithm `topological_sort(G)` that takes a graph G and returns an array $V = v_1, v_2, \ldots, v_n$ in which the verexes of G are sorted in topological order. This means that, for all pairs of vertexes v_i, v_j with $i < j$, there is no edge (v_j, v_i) in G. In other words, the vertexes of G are sorted in such a way that there are no backward edges. If G contains a cycle, then there is no topological order. In this case, `topological_sort(G)` must return `None`.