Teaching assistant: Malory Marin (LIP, MC2), ✉ malory.marin@ens-lyon.fr.
Some exercises are based on Chapter 22 of the book *Introduction to Algorithms*, see TD 1.

Data structures and algorithms for graphs (2h)

Exercise 1. ***Representations of graphs***

A graph is commonly given by its *adjacency matrix*. For algorithmic purposes, however, it is often more appropriate to represent a graph by its *adjacency list*, especially if the graph is sparse. This exercise explores these (and other) graph representations and their relationship.

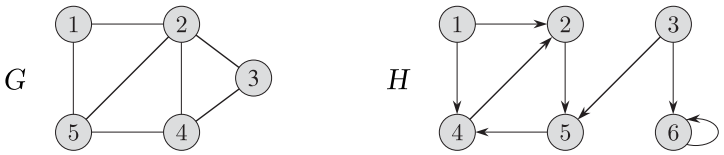
The **adjacency list** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the list $Adj[u]$ contains precisely those vertices $v \in V$ such that there is an edge $(u, v) \in E$. When writing pseudo-code, we consider adjacency lists as *attributes* of graphs, so we write $G.Adj$ to refer to an adjacency list Adj that represents a graph G.

Let us assume that the vertices of a graph $G = (V, E)$ are numbered with $1, 2, \ldots, |V|$ in an arbitrary way. The **adjacency matrix** of G is a $|V| \times |V|$ matrix $A = A(G) = (a_{i,j})$ defined as

$$a_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

(a) Give an adjacency list that represents the complete binary tree on seven vertices. Give an equivalent adjacency matrix. The vertices are numbered from 1 to 7 as in a binary heap.

(b) Write down the adjacency lists and adjacency matrices that represent the graphs below:



(c) Adapt the definitions of adjacency list and adjacency matrix to edge-weighted (di)graphs. To this end, we assume that the edge weights are given by a function $w\colon E \to \mathbb{R}$.

(d) Write down an algorithm (informally, or in pseudo-code) that takes an adjacency matrix of a (directed or undirected) graph G as input and computes its adjacency list.

(e) Write down an algorithm (informally, or in pseudo-code) that takes an adjacency list of a (directed or undirected) graph G as input and computes its adjacency matrix.

(f) It was mentioned above that using the adjacency list instead of the adjacency matrix to represent a sparse graph can be advantageous in certain situations. Why is this?

(g) Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

(h) The **incidence matrix** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{i,j})$ such that

$$b_{i,j} = \begin{cases} -1 & \text{if the edge } j \text{ leaves vertex } i, \\ 1 & \text{if the edge } j \text{ enters vertex } i, \\ 0 & \text{otherwise.} \end{cases}$$

What do the entries of the matrix product $BB^\top$ represent? ($B^\top$ is the *transpose* of B.)

Exercise 2. ***Extracting information from graph representations***

(a) Consider the following problem: *"Given two vertices i and j of a graph, decide if $(i, j) \in E$, i.e., if there is an edge from i to j."* What is the worst-case complexity of solving this problem, if the graph G is represented by an (i) adjacency list, (ii) adjacency matrix.

(b) The **transpose** of a directed graph $G = (V, E)$ is the graph $G^\top = (V, E^\top)$, where $E^\top = \{(u, v) \in V \times V : (v, u) \in E\}$. In words, $G^\top$ equals G with all its edges reversed. Describe efficient algorithms for computing $G^\top$ from G, for both the adjacency list and adjacency matrix representations of G. Analyze the running times of your algorithms.

(c) Most graph algorithms that take an adjacency-matrix representation as input require time $\Omega(|V|^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a **universal sink** (a vertex with in-degree $|V| - 1$ and out-degree 0) in time $O(V)$, given an adjacency matrix for G.

Exercise 3. ***Closeness centrality and Floyd-Warshall algorithm***

In a connected graph, the *normalized closeness centrality* (or closeness) of a node is the inverse of the average length of the shortest paths between the node and all other nodes in the graph. Thus, the more central a node is, the closer it is to all other nodes.

More formally, given a graph $G = (V, E)$, the closeness of a vertex $v \in V$ is:

$$C(v) = \frac{|V| - 1}{\sum_{u \in V} d(u, v)}$$

where $d(u, v)$ is the distance between u and v in G (i.e., the length of the shortest path between them).

For simplicity, assume that $V = \{1, \ldots, n\}$. For $i, j \in V$ and $1 \leqslant k \leqslant n$, let $P[i, j, k]$ be the length of the shortest path between i and j, using only the first k vertices (except i and j). When there is no such path, $P[i, j, k]$ is set to $+\infty$.

(a) Prove the following recurrence relation:

$$P[i, j, k] = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } i \neq j \text{ and } (i, j) \in E \\ \min(P[i, j, k-1], P[i, k, k-1] + P[k, j, k-1]) & \text{if } k \geqslant 1 \end{cases}$$

Hint: This recurrence relation corresponds to the logic of updating the shortest path between two vertices i and j by considering an intermediary vertex k.

(b) For $1 \leqslant i, j \leqslant n$, what is $P[i, j, n]$?

(c) Write a function that, given the adjacency matrix of a graph G, computes the shortest distances $d(i, j)$ for all $1 \leqslant i, j \leqslant n$ using the Floyd-Warshall algorithm.

(d) Write a function that, given the adjacency matrix of a graph G, computes the closeness centrality of all vertices.

(e) Using the NetworkX library, load the coappearance network of characters in the novel *Les Misérables*, compute the closeness centrality of each character, and plot the network with each node colored according to its closeness centrality.

Graph connectivity (2h)

Exercise 4. ***Giant component phenomenon***

In this exercise, you will investigate the emergence of the giant component in Erdős-Rényi random graphs. An Erdős-Rényi graph $G(n, p)$ is a graph with n vertices where each possible edge between any pair of vertices is included with independent probability p.

A *connected component* of a graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, there exists a path in G that connects u and v. In other words, all the vertices in a connected component are reachable from each other, and no vertex in C is connected to any vertex outside C.

(a) **Create Erdős-Rényi graphs:** Write a Python function that generates Erdős-Rényi graphs $G(n, p)$ using the `networkx` library.

(b) **Find the largest connected component:** Write a function that computes the size of the largest connected component in a given graph.

(c) **Simulate the giant component phenomenon:** For different values of the edge probability p, compute the size of the largest connected component and plot it against p.

(d) **Critical threshold:** The critical probability for the emergence of the giant component in an Erdős-Rényi graph is approximately $p_c = \frac{1}{n}$, where n is the number of nodes. Investigate how the size of the largest component evolves around this threshold.

(e) **Investigate the threshold for graph connectivity.** Write a function that simulates Erdős-Rényi graphs for different values of pp and checks whether the graph is fully connected (i.e., whether there is a path between every pair of vertices). Plot the fraction of connected graphs as a function of p. The critical probability for connectivity is approximately $p_{\text{conn}} = \frac{\log(n)}{n}$. Investigate how the probability of full connectivity evolves around this threshold and compare it to the emergence of the giant component.

Exercise 5. ***Depth-first search***

The goal of this exercise is to design an algorithm that will return the connected components of a graph G. The strategy followed by **depth-first search** (DFS) is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.

The procedure DFS colors vertices during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint. Besides creating a *depth-first forest*, DFS also timestamps each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v).

The procedure DFS below records when it discovers vertex u in the attribute $u.d$ and when it finishes vertex u in the attribute $u.f$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u, $u.d < u.f$. Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter. The following pseudocode is the basic depth-first-search algorithm. The input graph G may be undirected or directed. *time* is a global variable used for timestamping.

DFS(G)

```
1: for each vertex u ∈ G.V do
2:     u.color = WHITE
3:     u.π = NIL
4: end for
5: time = 0
6: for each vertex u ∈ G.V do
7:     if u.color == WHITE then
8:         DFS-VISIT(G, u)
9:     end if
10: end for
```

DFS-VISIT(G, u)

```
1: time = time + 1                    ▷ white u has just been discovered
2: u.d = time
3: u.color = GRAY
4: for each v ∈ G.Adj[u] do           ▷ explore edge (u, v)
5:     if v.color == WHITE then
6:         v.π = u
7:         DFS-VISIT(G, v)
8:     end if
9: end for
10: u.color = BLACK                   ▷ blacken u; it is finished
11: time = time + 1
12: u.f = time
```

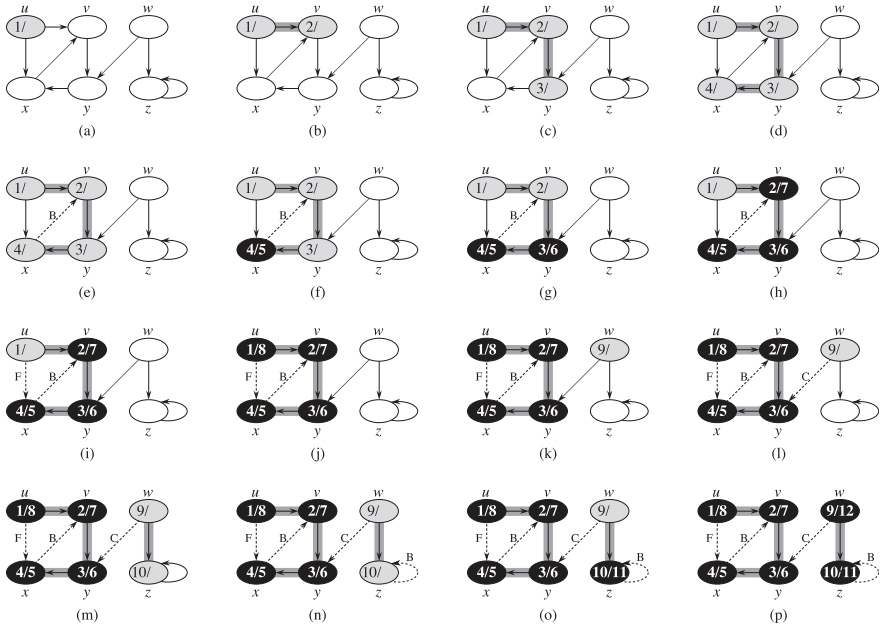


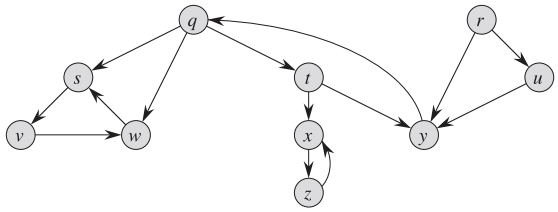
Figure 1: DFS on a directed graph. As edges are explored, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Non-tree edges are labeled B, C, or F according to whether they are *back*, *cross*, or *forward* edges. Timestamps within vertices indicate discovery / finishing times.

We can define four edge types in terms of the *depth-first forest* G_π produced by DFS on G:

1. ***Tree edges*** are edges in G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

(a) Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

(b) Show how DFS works on the graph below. Assume that the for loop of lines 6–10 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.



(c) *[*]* Show how DFS works on the graph above, if we turn it into an undirected graph, i.e., forget the arrows.

(d) Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure would produce the same result if line 3 of DFS-Visit was removed.

(e) Show that edge (u, v) is

- a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,
- a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and
- a cross edge if and only if $v.d < v.f < u.d < u.f$.

(f) Show that we can use DFS on an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.