1. Basic Graph theory and definitions:

Def 1: A **simple graph** G is a pair $G = (V, E)$ where V is a finite set, called the vertices (or nodes) of G, E is a subset of $P_2(V)$ (power set of size 2 of V), called the edges of G.

Example:

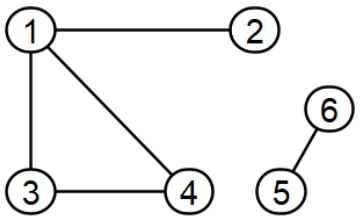

Figure 1 – A simple graph.

Let's consider the graph from Figure 1. This graph can be defined as $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{\{1,2\}, \{1,4\}, \{1,3\}, \{3,4\}, \{6,5\}\}$.

Note: the edges are subsets of 2 vertices, in an undirected simple graph, but to simplify the notation we will always use (v_1, v_2) to mark an edge in a simple graph, directed[1] or not. Thus, in our notation $E = \{(1,2), (1,4), (1,3), (3,4), (6,5)\}$.

Notations: **n** = |V| (we will call **n** the number of vertices)
m = |E| (we will call **m** the number of edges)

Obs.: graph vertices can have any label, but usually, by convention, they are labeled from 1 to n (or from 0 to n-1). We will use this convention for this course.

We say that 2 vertices are **adjacent** (or neighbors) if there is an edge between them (e.g. vertex 4 is adjacent to vertex 1 and to vertex 3).
For an edge (v_1, v_2), the vertices v_1, v_2 are called the **endpoints** of the edge (e.g. vertices 1 and 2 are the endpoints of edge (1,2)).
An edge is said to be **incident** to its endpoint vertices, also a vertex is said to be **incident** to the edges for which it is an endpoint (e.g. edge (1,2) is incident to vertices 1 and 2, while vertex 1 is incident to edges (1,2), (1,3) and (1,4)).
The number of edges a vertex is incident to is called the **degree** of the vertex (e.g. vertex 4 has degree 2 while vertex 1 has degree 3).

For a generic graph, both vertices and edges have a label:

Def 2: A **graph** is a triple $G = (V, E, \varphi)$ where:
- V is a finite set, called the vertices (or nodes) of G.
- E is a finite set, called the edges of G (also called the **labels** of the edges).
- φ is a function, $\varphi: E \rightarrow P_2(V)$.

For example, let's consider the graph in Figure 1 but add labels to edges:

[1] See part 2B for directed graph definitions

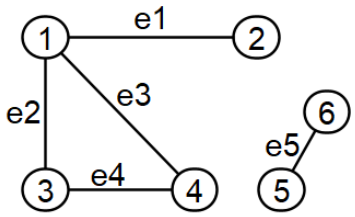

Figure 2 – Graph from Figure 1 with labels

The graph from Figure 2 can be defined as $G = (V, E, \varphi)$, where V = {1,2,3,4,5,6}, E = {e1, e2, e3, e4, e5} and φ = {e1: (1,2), e2: (1,3), e3: (1,4), e4: (3,4), e5: (5,6)}.

Note: this graph still is a simple graph, just with a different (more general definition).

Def 3: a **subgraph** of graph $G = (V, E)$ is a graph $G'=(V', E')$ such that $V' \subseteq V$, $V' \neq \emptyset$ and $E' \subseteq E$.
So, a subgraph is a graph that contains some or all vertices from the original graph (at least 1) and some or all of the edges of the original graph.

2. Major types of graphs

A) Multigraph

A **multigraph** is a graph that can have more than one edge joining the same endpoints or edges for which the same vertex is both endpoints (i.e. self-loops).

Figure 3 – a multigraph

In Figure 3 you can see a multigraph example. edge e1 is a self-loop (both endpoints are vertex 1) while edges e3 and e4 have the same endpoints (2 and 3).
Note that since multiple edges can have the same endpoints, a multigraph can be defined only with the general definition – $G = (V, E, \varphi)$ – to be able to differentiate between those edges. During this course we will generally work with simple graphs.
To be clear: a graph is "simple" if it not a multigraph.

B) Directed graph

A graph $G=(V, E)$ is a **directed graph** if E is a subset of VxV (instead of $P_2(V)$). That means that the edges have an *orientation,* so the edge (v_1, v_2) is different from the edge (v_2, v_1). A graph that is not directed is called **undirected**.

Let (v_1, v_2) be an edge in a directed graph. v_1 is called the **initial** (or starting) vertex and v_2 is called the **terminal** vertex. Edge (v_1, v_2) is called an **outbound** edge of v_1 and an **inbound** edge of v_2. v_2 is an *outbound neighbour* of v_1, while v_1 is an *inbound neighbor* of v_2.

The number of *inbound* edges of a vertex is called the **indegree** while the number of *outbound* edges is called the **outdegree**. If d_T is the (total) degree of a vertex, d_{IN} is the indegree and d_{OUT} is the outdegree then:
$d_T = d_{IN} + d_{OUT}$

Note that if a directed graph contains an edge in both directions for the same endpoints (i.e. for some vertices v_1 and v_2 the graph contains both (v_1, v_2) and (v_2, v_1)) the graph is **not** a multigraph.
In Figure 4 is show an example of a directed graph.

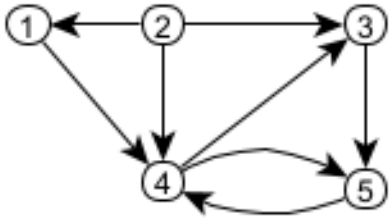

Figure 4 – a directed graph

The graph in figure 4 is a directed graph $G = (V, E)$, where $V = \{1,2,3,4,5\}$ and $E = \{(1,4), (2,1), (2,3), (2,4), (3,5), (4,3), (4,5), (5,4)\}$. For this graph, node 4 has 2 *outbound* edges – (4,3), (4,5) – and 3 *inbound* edges – (1,4), (2,4), (5,4). The outbound neighbors of vertex 4 are {3,5} and the inbound neighbors are {1,2,5}. The outdegree of vertex 4 is 2 while the indegree is 3.

C) Weighted graph

A weighted graph is a graph that has a number (called weight) associated to its vertices (called vertex-weighted graph) and/or edges (called edge-weighted graph). For this curse we will only consider edge-weighted graphs.
Formal definition:
A **weighted graph** $G = (V,E,w)$ is a graph $G=(V,E)$ to which a function $w:E\rightarrow \mathbb{R}$ was associated, called the weighting function. In Figure 5 an example weighted graph is given.

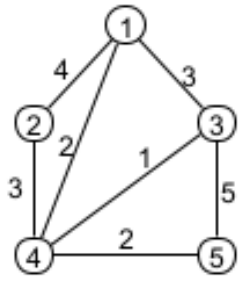

Figure 5 – a weighted graph

The graph from Figure 5 can be defined as $G = (V, E, w)$, where $V = \{1,2,3,4,5\}$, $E = \{(1,2), (1,3), (1,4), (2,4), (3,4), (3,5), (4,5)\}$ and $w = \{(1,2):4, (1,3):3, (1,4):2, (2,4):3, (3,4):1, (3,5):5, (4,5):2\}$.

It is important to note that all these types of graphs can be mixed and matched, for example you can have a weighted directed graph or a directed multigraph etc. all of them with or without edge labels (unless they are multigraphs, then edge labels are required).

3. Representations

Until now we have discussed what a graph is conceptually. Now we will talk about how to practically implement a graph.

From a programming perspective, a graph is an Abstract Data Type (ADT) – in other words we know what operations a graph has but not their implementation. Indeed, the implementation for the same graph operation can differ, depending on what data structures and algorithms we use. The data structures we chose and how we use them is called the **representation** of the graph. In general, each representation has advantages and disadvantages. To compare them we will consider the extra space needed for the representation and the time complexity for some graph operations.

Graph operations we will use for comparison:

- `is_edge(v1, v2)` – checks if an edge exists between 2 vertices. If the graph if the graph is directed, then it checks only for the given order.
- `get_neighbors(v)` – returns a list with the neighbors of the given vertex. If the graph is directed, then it returns a list with only the outbound neighbors.
- `get_inbound_neighbors(v)` – if the graph is directed then returns a list with the inbound neighbors of the graph, otherwise it is the same as `get_neighbors`.

Note that actual graph implementations will not return a list with a copy of the neighbors, but only an iterator over the neighbors, which is more efficient. But to highlight that we want to measure the complexity of iterating over all the neighbors of a vertex, we will consider that the function creates a new list and adds all the neighbors to it, therefore the function doing the iteration. For exemplification of the representations, let's consider the graph from figure 4 (copied here for easier reference).

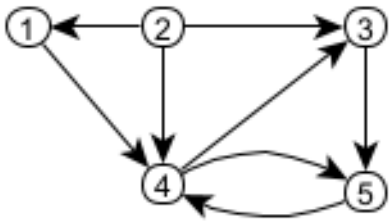

Figure 4 – a directed graph

A) List of edges and vertices

The easiest way to represent a graph is just to have a list of all edges in a graph, and a list of all vertices. If we have a convention that all vertices are named 0 to n-1 or 1 to n, then we do not even need the list of vertices, just the number of vertices (n).

The graph in figure 4 represented as a list of edges and vertices is:

edges = [(1,4), (2,1), (2,3), (2,4), (3,5), (4,3), (4,5), (5,4)], n = 5

In this representation, doing any operations means going through the list of all edges.

The space needed to represent this graph is m, since we just need the list of edges.

- `is_edge(v1, v2)` is in O(m) since we need to search through the list until we find it.
- `get_neighbors(v)` and `get_inbound_neighbors(v)` are both in Theta(m) since we need to check the entire list of edges, no matter what, to make sure we find all the neighbours.

If the list of edges is sorted, all 3 can be done in O(log(m) + n) in worst case and O(log(m) + m/n) on average.

B) Adjacency matrix

This representation uses a matrix of size n x n where value on line i and column j is 1 if there is an edge from vertex i to vertex j, and 0 (or null) otherwise.

The adjacency matrix for the graph in figure 4 is:

	1	**2**	**3**	**4**	**5**
1	0	0	0	1	0
2	1	0	1	1	0
3	0	0	0	0	1
4	0	0	1	0	1
5	0	0	0	1	0

Note that an adjacency matrix for an undirected graph must be symmetrical, since undirected means that if there is edge (i,j) then there is also edge (j, i).

Since we need a matrix with size n x n then the space needed will be n^2.

- `is_edge(v1, v2)` is in Theta(1) since all we need to do is check a value in a matrix at given indices.
- `get_neighbors(v)` is in Theta(n): to check all the (outbound) neighbors we need to go over the entire line for that vertex, which has n elements.
  For example, for vertex 2 we need to check line 2 to see at which columns we have values of 1, and we find columns 1,3, and 4, which are then the outbound neighbors of vertex 2.
- `get_inbound_neighbors(v)` is also in Theta(n), it is the same as `get_neighbors` but searches on the column instead of the line.
  For example, for vertex 3 we need to check column 3 to see at which lines we have values of 1, and we find lines 2 and 4, which are then the inbound neighbors of vertex 3.

C) Incidence matrix

This representation uses a matrix with size n x m, where at index i,j we have 1 if vertex i is incident to edge j or 0 otherwise. In the case of directed graphs, we have 1 if the vertex is the initial vertex of the edge j, 2 if it is the terminal vertex of the edge and 0 otherwise.

The notation with 1 and 2 used for directed matrices is just a convention used during this course, there is no universal convention.

The incidence matrix of the graph in figure 4 is the following:

	(1,4) e1	**(2,1) e2**	**(2,3) e3**	**(2,4) e4**	**(3,5) e5**	**(4,3) e6**	**(4,5) e7**	**(5,4) e8**
1	1	2	0	0	0	0	0	0
2	0	1	1	1	0	0	0	0
3	0	0	2	0	1	2	0	0
4	2	0	0	2	0	1	1	2
5	0	0	0	0	2	0	2	1

Usually, an incidence matrix is used with labeled edges, for simplicity we did not label the edges in the figure, but added labels in the incidence matrix.

Since we use a matrix of size n x m then the space needed is n*m.

- `is_edge(`v_1`, `v_2`)` is in O(m): First we need to search on the lines of vertices v_1 and v_2 at the same time to find a column where both values are different from 0.
  For example, to find if there is an edge between vertices 2 and 4 we search the lines 2 and 4 at the same time: we look first at the value 2,e1 (2) and 4,e1 (0), since one of them is 0, we go to the next column, we check 2,e2 (0) and 4,e2 (0), since one of them is zero we go to next and so on until we find a column were both are different from 0: 2,e4 is 1 and 4,e4 is 2, both of them are different from 0 that means that edge e4 is an edge between 2 and 4.
  For directed graphs we also need to make sure that the values 1 and 2 are in the correct order.
- `get_neighbors(v)` is in O(n*m): to check all the (outbound) neighbors we need to go over the entire line for that vertex, and where we find a value of 1 to check that entire column to find the corresponding vertex.
  For example, for vertex 2 we need to check line 2 to see at which columns we have values of 1, and we find columns e2, e3, and e4. Then we search in column e2 for the line with value 2, we find on line 1, thus 1 is an outbound neighbor of vertex 2. We do the same for columns e3 and e4 and we find vertices 3 and 4.
- `get_inbound_neighbors(v)` is also in O(n*m), it is the same as `get_neighbors` but we search initially for values of 2, and then we search for values of 1 on the column (reverse 1 and 2 from `get_neighbors`).
  For example, for vertex 3 we need to check line 3 to see at which lines we have values of 2, and we find lines e3 and e6. Then we search in column e3 for the line with value 1, we find on line 2, thus 2 is an inbound neighbor of vertex 3. We do the same for column e6 and find the other inbound neighbor: 4.

D) (Double) List of neighbors (adjacency list)

The idea for this representation is to have a list of all the neighbors for each vertex. Usually, the link between a vertex and its list of neighbors is made through a map (dictionary). For directed graph, the list of neighbors contains just the outbound neighbors. We can add *another* map containing a list with all the inbound neighbors, in which case the structure is called a *double* list of neighbors (since we have 2 lists for each vertex).

Note that using a double list of neighbors for undirected graphs does not make much sense (since both lists should be equal).

The (double) list of neighbors for our example is:

list_of_neighbors = {1: [4], 2: [1,3,4], 3: [5], 4: [3,5], 5: [4]}

(list_of_inbound_neighbors = {1: [2], 2: [], 3: [2,4], 4: [1,2,5], 5: [3,4]})

For space complexity we need a list for each vertex, and all lists will represent all edges, so in total we use n+m space. If we use a double list of neighbors the we need 2(n+m).

- `is_edge(`v_1`, `v_2`).` For this we need to check the list of neighbors of v_1. In the worst case, if v_1 is incident to all the edges, this is O(m). But this rarely will happen, and on average this is O(m/n), since m/n is the average number of edges per vertex.
- `get_neighbors(v).` For this we need to copy the list of neighbors. As before, the list of neighbors is O(m) in the worst case, but O(m/n) in the average case.
- `get_inbound_neighbors(v).` Here is where it matters if we have a double list of neighbors or not. Without the inbound neighbors list, we have to search the entire list of neighbors to see for which neighbor our given vertex is an outbound neighbor. And that is O(m+n). If we have an inbound list of neighbors then we just need to copy it, which is, on average O(m/n).

Summary comparison:

Representation	**Space needed**	**Time complexity**		
		`is_edge`	`get_neighbors`	`get_inbound_neighbors`
List of edges	m	O(m)	Theta(m)	Theta(m)
Adjacency Matrix	n^2	Theta(1)	Theta(n)	Theta(n)
Incidence matrix	n*m	O(m)	O(n*m)	O(n*m)
List of neighbors	n + m	O(m/n)	O(m/n)	O(m + n)
Double list of neighbors	2(n + m)	O(m/n)	O(m/n)	O(m/n)

Notes:

In general, the most efficient representation is the list of neighbors (also called adjacency list). The double list of neighbors is only used for directed graphs if we know we will need the inbound neighbors.

Adjacency matrix is efficient in checking if an edge exists, so if we run algorithms on the graph that need many checks for edges then an adjacency matrix is recommended. Also,

when the graph is **dense** (i.e. there are many edges, m approaches n^2) then the difference between Theta(n) and O(m/n) is not as significant so an adjacency matrix may be more efficient than a list of neighbors.

The incidence matrix is an inefficient representation rarely used, but it has its purpose for some specific problems, such as incidence coloring ; but it also make much more sense for hypergraphs.

4. Walks and derivates

Walk: A walk between two vertices *s* and *t* of a graph *G=(V, E)* is an alternating sequence of vertices and edges between these two vertices (i.e. *s* e_1 v_1 e_2 v_2 .... e_L *t)* such that for any triplet (vertex edge vertex) in the walk, the 2 vertices are the endpoints of the edge. The vertex *s* is called the *initial* or *starting* vertex and *t* is called the *terminal* vertex of the walk.

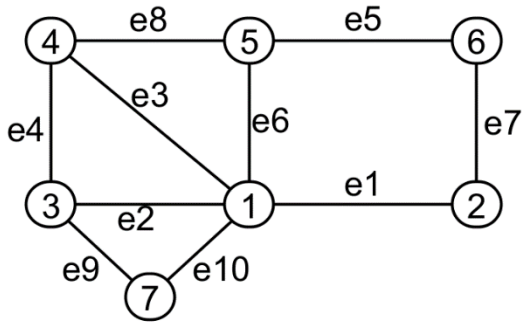

Figure 6 – an example graph

Let's consider the example in Figure 6. A walk in this graph might be: (1 e6 5 e8 4 e3 1 e6 5 e5 6 e7 2). The starting vertex of this walk is 1 while the terminal vertex of the walk is 2.

The **length** of a walk is the number of edges it contains. The walk in the previous example has a length of 6 (edges: e6, e8, e3, e6, e5, e7 – 6 edges in total).

While a walk is defined as a sequence of vertices and edges, for simplicity we will most often not use all elements in a walk. Indeed, for a simple graph, a sequence of vertices accurately describes any walk, while for both simple graphs and multigraphs a sequence of edges accurately describes any walk with length higher than 1. Also, most of the time we will not label edges, so we might have to use pairs of vertices to refer to edges, even when describing a walk.

So, the previous example walk can also be written as:

- without labels: 1 (1,5) 5 (5,4) 4 (4,1) 1 (1,5) 5 (5,6) 6 (6,2) 2
- only vertices: 1 5 4 1 5 6 2
- only edges: e6 e8 e3 e6 e5 e7
- only edges without labels: (1,5) (5,4) (4,1) (1,5) (5,6) (6,2)

Most of the time we will use "only vertices" as it is the most concise, but we will use "only edges" (usually without labels) when dealing with trails and circuits (where what edges are used matter).

A walk for which *s=t* (starts and ends at the same vertex) is called a *closed walk*. A walk that is not closed is an *open walk*.
In a directed graph, a walk must obey the direction of the edges. That is, for any triple (vertex1 edge vertex2) in the walk, vertex1 must be the initial vertex of the edge while vertex2 must be the terminal vertex of the edge.

A **trail** is a walk with no repeated edges. A trail in the graph at Figure 6 might be: (3,1) (1,5), (5,4), (4,1), (1,7).
A **path** is a walk with no repeated vertices (except the initial and terminal vertices). A path in the graph at Figure 6 might be: 7 3 4 1 2 6.

Note that, in order to repeat an edge in a walk, the endpoints of that edge must be repeated too. Therefore, if we have no repeated vertices there cannot be any repeated edges.

In other words: any path is also a trail. But not any trail is also a path. In the given example trail, while there are no repeated edges, vertex 1 is repeated.

A **trail** that starts and ends at the same vertex is called a **circuit**. An example circuit for the graph in Figure 6 might be: (1,4) (4,3) (3,1) (1,5) (5,6) (6,2) (2,1).

A **path** that starts and ends at the same vertex is called a **cycle**. An example cycle for the graph in Figure 6 might be: 1 3 4 5 6 2 1.

A graph is said to be **connected** if there is a path between any 2 vertices of the graph. If a graph is not connected it is called **disconnected.**
For example, the graph in Figure 1 is disconnected while the graph in figure 6 is connected.