1. Subgraphs:

Reminder: Let $G = (V, E)$ be a simple graph. $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$, $V' \neq \emptyset$ and $E' \subseteq E$.

Types of subgraphs:

G' is called an **induced** subgraph if $\forall\ v_1, v_2 \in V'$, if $(v_1, v_2) \in E$ then $(v_1, v_2) \in E'$.
In other words, E' contains all edges from E which have the vertices from V' as endpoints.

G' is called a **spanning** subgraph if $V'=V$.
In other words, a spanning subgraph is a subgraph that has all the vertices but only some of the edges.

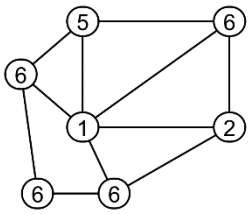

Figure 1

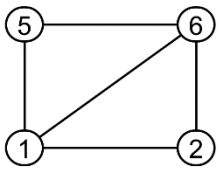

Figure 2

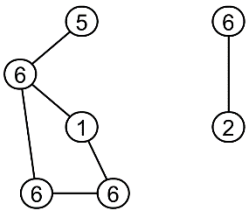

Figure 3

Consider the graphs in Figures 1, 2 and 3. The graph in Figure 2 is an *induced* subgraph of the graph in Figure 1, while the graph in Figure 3 is a *spanning* subgraph of the graph in Figure 1.

2. Traversals

Traversing the graph refers to going through each vertex of the graph once, from a given starting vertex. If the graph is disconnected[1] not all vertices of the graph might be reachable from the chosen starting vertex.

There are 2 main basic ways for that: **Breadth first search** (BFS – not to be confused with Best First Search) and **Depth first search** (DFS).

In BFS the idea is that we visit in levels: all neighbors first then all neighbors of the first set of neighbors, then their neighbors and so on.

Take for example the Graph in Figure 4. If the starting vertex is 4, then The BFS traversal will start with vertex 4. Then it will go for the neighbors of vertex 4: 3, 1 and 5. Then it will go for the neighbors of the neighbors, that is all the neighbors of 3, all the neighbors of 1 and all the neighbors of 5 (naturally ignoring vertices that were already visited): 6, 2 and 7. If there were

[1] See part 3

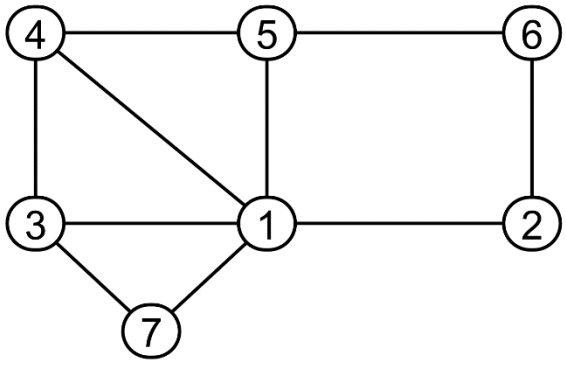

Figure 4

more vertices, it then would go for all neighbors of 6,2,7 and so on and so forth. While the traversal is defined by this level-like visiting of neighbors, the order in which it visits the neighbors on a specific level is not specified – it can be any order, thus there can be multiple different BFS traversals for the same graph and starting vertex.
For our example, starting from vertex 4, a valid traversal might be: 4, 5, 1, 3, 6, 2, 7.

BFS is an especially useful traversal since the level on which we find a certain vertex is equal to the length of the shortest path from the start to that vertex. Thus, a slightly modified BFS traversal can be used to find the shortest path from one vertex to another vertex.

The idea in DFS is that it goes for the first neighbor of the start, then the first neighbor of that vertex and so on until it reaches a vertex with no unvisited neighbors, at which point is "backtracks" to the most recently visited vertex which still has unvisited neighbors and continues from there.

In our example, if we start from vertex 4, The DFS might go 4, 5, 1, 3, 7 at which point it finds that 7 does not have any unvisited neighbors, so it "backtracks" to vertex 5, which is the most recent vertex which still has unvisited neighbors, then it continues 6, 2.

So, a DFS traversal might be 4, 5, 1, 3, 7, 6, 2. Obviously this is not the only possible DFS traversal, the order in which the neighbor is chosen is fixed so there might be many possible DFS traversals.

The algorithms for BFS and DFS are very similar. In fact they may be implemented the same way except for the fact that one uses a queue (BFS) and another a stack (DFS).

In the following a BFS/DFS algorithm is given. This is a simple algorithm that just prints the vertices in a traversal, and therefore is not really useful standalone. But this algorithm is the base for any algorithm that uses BFS or DFS (all graph algorithm that want to traverse the graph for a reason or another, and there are many such algorithms), as well as the base for BFS-based finding of the shortest path and length.

```
Algorithm name: BFS/DFS
Input:
	g: a graph
	s: a starting vertex in g
Output:
	prints the vertices in a BFS/DFS traversal of g starting from s
Algorithm:
	Queue/Stack d  #Queue used for BFS, stack used for DFS
	Set visited
	d.push(s)
	while d is not empty:
		current = d.pop()
		print(current)
		for neighbor in g.get_neighbors(current):
			if neighbor not in visited:
				d.push(neighbor)
			end_if
		end_for
	end_while
```

3. Connectivity

Reminder:

- a **walk** is a sequence of alternating vertices and edges such that the edges are between their 2 endpoints (or, more simply, a sequence of vertices such that any 2 consecutive vertices are adjacent)
- a **trail** is a walk with no duplicate edges. A **circuit** is a trail that starts and ends at the same vertex.
- a **path** is a walk with no duplicate vertices. A **cycle** is a path that starts and ends at the same vertex.

A graph is said to be **connected** if there exists a path between any pair of its vertices. If a graph is not connected it is said that it is **disconnected**.

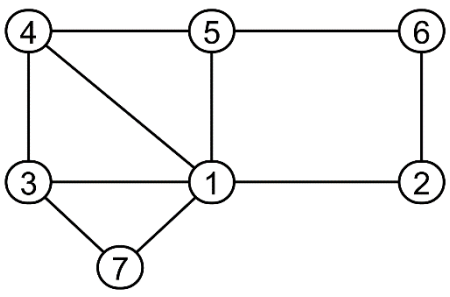

Figure 5

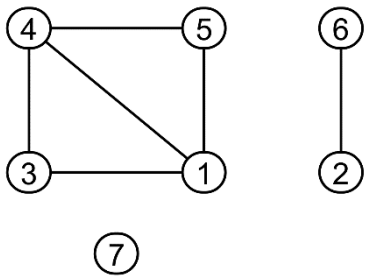

Figure 6

For example, consider the graphs in Figure 5 (same as Figure 4) and Figure 6. The graph in Figure 5 is a connected graph. The graph in Figure 6 is a disconnected graph.

For any graph, it is possible to partition it into subgraphs in such a way that each subgraph is connected, and these subgraphs are not connected between them. Each such subgraph is called a **connected component** of the graph.

For example, the graph in figure 6 has 3 connected components: the subgraph induced by vertices {4, 5, 3, 1}, the subgraph induced by {2, 6} and the graph containing vertex 7.

Observation: a graph with one connected component is a connected graph.

How to check if a graph is connected? – making a traversal (BFS/DFS) from any random node and checking if all the nodes have been traversed.

For example, doing a BFS traversal in graph in Figure 5, starting from node 4, will result in traversing nodes 4,3,1,5,7, 2,6. Since these are all the nodes then that graph is connected.
On the other hand, if we do a BFS for the graph in Figure 6, starting from the node 4, then we will traverse vertices 4, 3, 1, 5. Since these are not all the nodes, the graph in Figure 6 is not connected.

How to get all the connected components? – starting a traversal from a random vertex, all the found vertices are part of a connected component. If they are not all the nodes, we do another traversal starting from a node not yet traversed. The nodes found are part of another connected component. And so on until all the nodes have been traversed.

For example, for the graph in Figure 6 we can do a traversal from node 1 and find vertices {1,3,4,5}. These are part of one directed component. Then we can do a traversal from node 7 and we find only {7}, another connected component. We then start another traversal from 2 and we find {2,6}. Now all vertices are found, and the connected components are the subgraphs induced by {1,3,4,5}, {7} and {2,6}.

Directed graphs – Strongly connected components.

For directed graphs the relevant element for connectivity is the Strongly Connected Component (SCC). Analyzing simple connectivity and simple connected components for directed graphs can be done, but for that we will just ignore the orientation of the edges (basically we just treat the graph as if an undirected graph).

A digraph is said to be **strongly connected** if there exists a path in both directions between any pair of its vertices. That is, $\forall\ v_1, v_2 \in V$, there is a path from v_1 to v_2 and a path from v_2 to v_1.
For example, consider the graph in Figure 7. While it is clear the graph is connected, the graph is not *strongly* connected, as, for example, there is no path from 9 to 1.

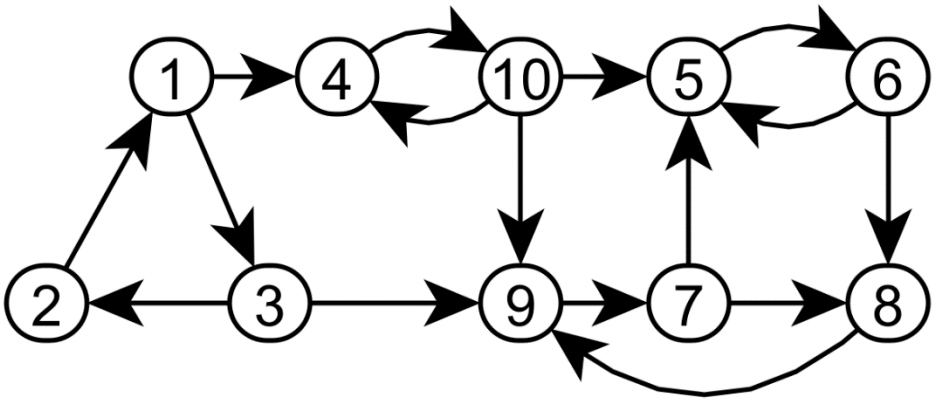

Figure 7

If G=(V, E) is a directed graph, subgraph G'=(V',E') of G is called a **Strongly Connected Component** (SCC) of G if:

a) G' is strongly connected.

b) ∄ v ∈ V so that if v and all its incident edges from E are added to G', G' is still strongly connected.

In the graph from Figure 7 there are 3 separate strongly connected components. They are highlighted in Figure 8.

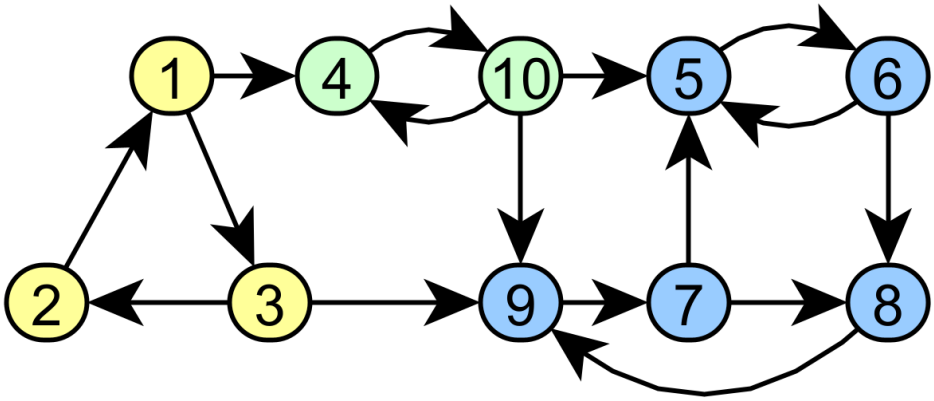

Figure 8 – Graph from Figure 7 with SCCs highlighted.

How to check if a directed graph is strongly connected? – Get all vertices with a traversal from a random vertex then get all vertices with a traversal from the same vertex on reverse graph (i.e. same vertices and same edges, but edges have opposite orientation). If the 2 sets of vertices are the same, then the graph is strongly connected.

Finding strongly connected components (efficiently) is more difficult, we will not be reviewing those algorithms. If you are curious you can check Tarjan's algorithm or Kosaraju's algorithm.

4. Notions related to walks in graph

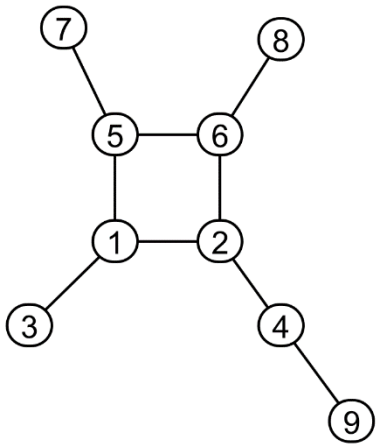

Figure 9

For each notion we will give an example from graph in Figure 9.

The **length** of a walk, a trail, a path, or a cycle is its number of *edges*. Thus, a path of n vertices has length n − 1, and a cycle of n vertices has length n.

For example. the walk 3,1,2,6,5,7 has length 5.

The **distance** between 2 vertices is the length of a shortest path between them. We note as **d(u,v)** the distance of between vertices u and v (starting in u ending in v, order matters for directed graphs).

For example, d(3,7) is 3, since the shortest path is 3,1,5,7.

The **eccentricity** of a vertex u in G is $\max_{v \in V(G)} d(u, v)$ and is denoted by ε(u). The eccentricity the length of the longest possible shortest path from this vertex to another vertex. Or, in other words, eccentricity of a vertex represents the distance from this vertex to the vertex that is farthest away.

For example, ε(3) is 4, as vertices 9 and 8 are the farthest away from 3, and d(3,9) = d(3,8) = 4.

The **radius** of a graph is the minimum eccentricity of any vertex in the graph.
The **center** of a graph is the set of vertices with minimum eccentricity.

For example, the radius of the graph is 3, as that is the samllest eccentricity of any node. Nodes 1, 2, and 6 have eccentricity 3. Then the center of the graph is {1,2,6}.

The **diameter** of a graph is the maximum eccentricity.

For example, for our graph the diameter is 5, as d(9,7)=5 so ε(9) is 5 (there is no other vertex with higher eccentricity in the graph.