Practical title: Graph Topology Analysis in Python.
Learning outcomes: Learn to construct and analyse different graph topologies.
Software requirements: Python 3
networkx (https://networkx.github.io/documentation/stable/)
matplotlib (https://matplotlib.org/)
Module: CMT202
Lecturer: Padraig Corcoran

To install the networkx and matplotlib libraries in a pipenv virtual environment use the following commands:
pipenv install networkx
pipenv install matplotlib

To run the Python program network_1.py pipenv virtual environment use the following command:
pipenv run python network_1.py

If you are unsure about how to istall libraries and run python programs in a pipenv virtual environment, please consult the MapReduce practical session.

Part 1 - Creating a graph using NetworkX
The topology of a distributed system can be modelled using a graph. A graph is a pair G=(V, E), where V is a set whose elements are called vertices (singular: vertex), and E is a set of pairs of vertices, whose elements are called edges. An example of a graph with six vertices and seven edges is displayed in Figure 1. In the example V = {1, 2, 3, 4, 5, 6} and E = {[1,2], [1,5], [2,3], [2,5], [3,4], [4,5], [4,6]}. Please read the following Wikipedia article for more info https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
In a distributed system vertices correspond to computers and edges correspond to communication links.

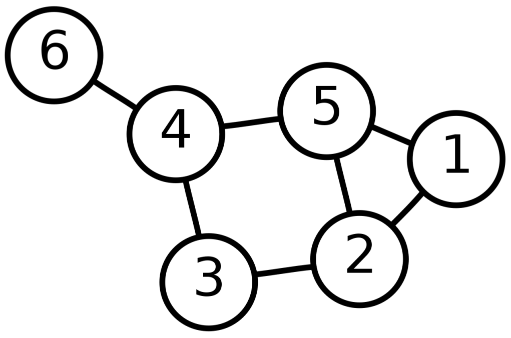


Figure 1. A graph with six vertices and seven edges.

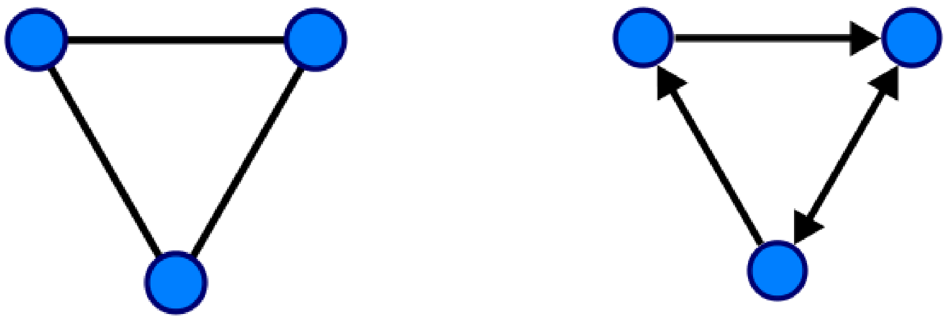
NetworkX is a Python library for graph analysis. The following Python program creates the graph in Figure 1. The method *nx.Graph()* creates the graph, the method *add_node(.)* adds a single vertex, the method *add_edge(.)* adds a single edge and the method *nx.draw(.)* draws the graph.

```
import networkx as nx
import matplotlib.pyplot as plt

# Create empty
G = nx.Graph()

# Add graph vertices
G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)
G.add_node(6)

# Add graph edges
G.add_edge(1, 2)
G.add_edge(1, 5)
G.add_edge(2, 3)
G.add_edge(2, 5)
G.add_edge(3, 4)
G.add_edge(4, 5)
G.add_edge(4, 6)

# Print graph vertices and edges
print("Graph vertices: ", G.nodes())
print("Graph edges: ",G.edges())

# Draw the graph
nx.draw(G, with_labels=True, pos=nx.spectral_layout(G))
plt.show()
```

network_1.py

Part 2 - Directed and undirected graphs

There are two main types of graphs: directed and undirected. In an undirected graph edges do not have a direction. While in a directed graph edges have a direction. Examples of directed and undirected graphs are displayed in Figure 2.



(a) (b)

Figure 2. Undirected and directed graphs are displayed in (a) and (b) respectively.

In the previous program (network_1.py) you created an undirected graph using the method nx.Graph(). In the following program you will create a directed graph using the method nx.DiGraph(). Notice that when you draw the graph the edges now have a direction.

```
import networkx as nx
import matplotlib.pyplot as plt

# Create empty
G = nx.DiGraph()

# Add graph vertices
G.add_nodes_from([1, 2, 3, 4, 5, 6])

# Add graph edges
G.add_edges_from([(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6)])

# Print graph vertices and edges
print("Graph vertices: ", G.nodes())
print("Graph edges: ",G.edges())

# Draw the graph
nx.draw(G, with_labels=True, pos=nx.spectral_layout(G))
plt.show()
```

network_2.py

Part 3: Searching in a Graph

The task of searching in a graph is to find a path from a specified start vertex to a specified target vertex. Consider the graph in Figure 3; a path from vertex 2 to vertex 6 is the list of vertices 2, 3, 4, 6.

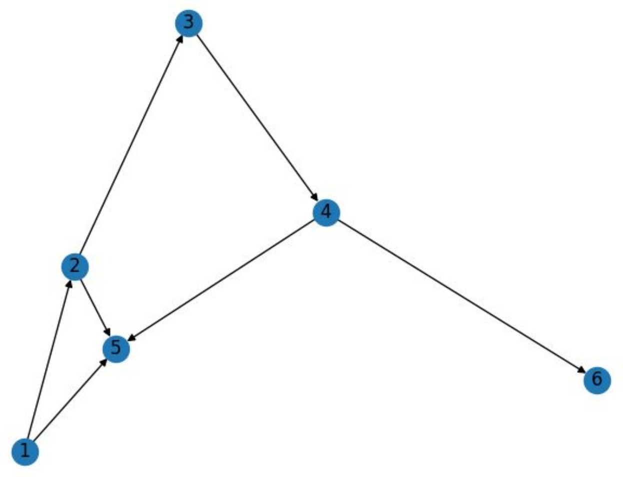


Figure 3. A directed graph.

There may be none or more than one path between two vertices in a graph. Identify examples to both these cases in the graph of Figure 3.
In most cases we are interested in finding the shortest path between two vertices. We can find the shortest between two vertices using the networkx *nx.shortest_path(.)* method. The following Python program uses this method to find the shortest path from vertex 2 to vertex 6 in the graph of Figure 3.

```
import networkx as nx
import matplotlib.pyplot as plt

# Create empty
G = nx.DiGraph()

# Add graph vertices
G.add_nodes_from([1, 2, 3, 4, 5, 6])

# Add graph edges
G.add_edges_from([(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6)])

# Print graph vertices and edges
print("Graph vertices: ", G.nodes())
print("Graph edges: ",G.edges())

# FInd shortest path from vertex 2 to vertex 6
p = nx.shortest_path(G, source=2, target=6)
print("Shortest path:", p)

# Draw the graph
nx.draw(G, with_labels=True, pos=nx.spectral_layout(G))
plt.show()
```

network_3.py

Part 4: Searching in structured P2P using Chord
Recall from the lecture notes entitled Architectures, that Chord is an algorithm for searching in a structured P2P. In this algorithm the P2P system is constructed to have a ring topology with additional shortcuts. Recall that, in a ring topology the graph has a circle shape.
Consider the graph in Figure 4. Note that the vertices in this graph are {1, 4, 9, 11, 14, 18, 20, 21, 28} and the edges are {(1, 4), (4, 9), (9, 11), (11,14), (14, 18), (18, 20), (20, 21), (21, 28), (28, 1), (1,9), (1, 18), (4, 20)., (9,14), (9, 18), (9, 28), (11, 28), (11, 20), (11,28), (14,28), (14,1), (18, 28), (18, 4), (20, 28), (20,4), (21,1), (21,9), (28, 14), (28,4)}.

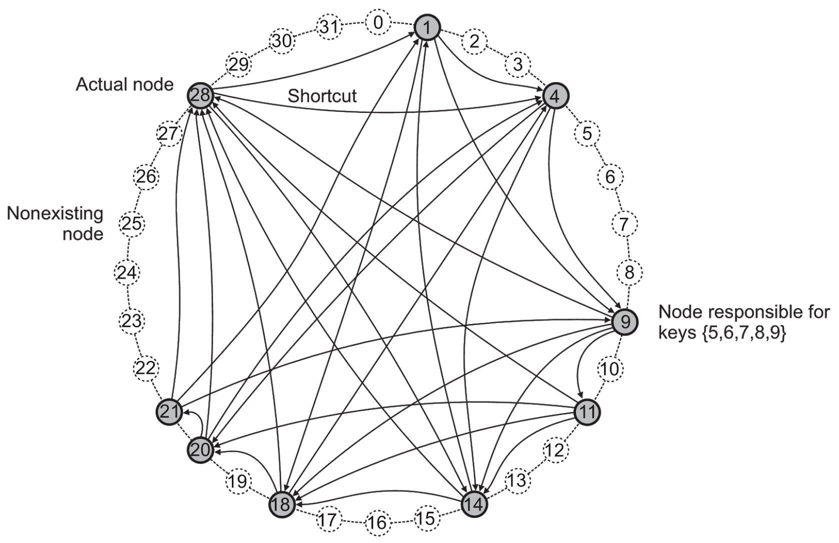


Figure 4. A ring topology with additional shortcuts.

Construct a graph in NetworkX containing only the edges corresponding to the ring topology; that is, the edges {(1, 4), (4, 9), (9, 11), (11,14), (14, 18), (18, 20), (20, 21), (21, 28), (28, 1)}. Perform an analysis of how the lengths of shortest paths changes as you introduce shortcuts. For example, when you add the edge (11,18) how do the lengths of shortest paths change?