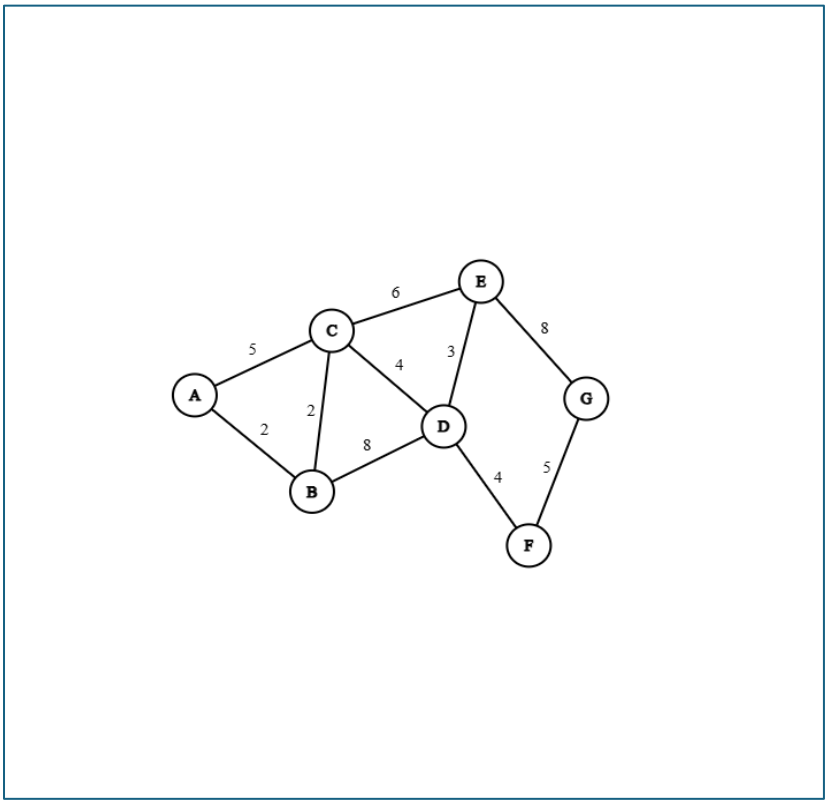
RELAZIONE PROGETTO – SIMULAZIONE PROTOCOLLO DI ROUTING

GABRIELE FRONZONI (matr. 0001068902)

Obiettivi:

- Realizzazione di uno script Python che simuli il popolamento delle tabelle di routing utilizzando l'algoritmo del Routing Distance Vector;
- Mostrare per ogni iterazione dell'algoritmo le tabelle di routing di ogni nodo. Le tabelle di routing di ogni nodo mostrano i nodi raggiungibili con relative distanze e next hop;

Per la simulazione della rete si è usato il seguente grafo, non orientato e pesato:



Per prima cosa si è definita una classe **Node** che simula un nodo di rete:

```
class Node:
    """
    A class that represent a network node.
    Each node has a Routing Table that stores, for each node, distances and next hop
    for each node.

    Attributes:
        name : str
            Name of the node.
        routingTable : dict
            A dictionary where keys are names of other nodes and values are pair of distance and
            name of the next hop.

    """
```

Ogni nodo ha come attributi il *nome* e la propria *Routing Table*, che viene memorizzata come un dizionario nel quale le chiavi sono i nomi degli altri nodi e il valore è una coppia composta dalla distanza da quel nodo (metrica) e dal next hop.

L'inizializzazione del nodo viene gestita dalla seguente funzione:

```
def __init__(self, name):
    """
    Initialize a node with its name and an initial routing table containing itself.

    Args:
        name : str
            The name of the node.
    """
    self.name = name
    self.routingTable = {name : (0, name) }
```

Mediante questa funzione viene settato il nome del nodo e viene inizializzata la tabella di routing con l'unica rotta che è quella verso sé stesso.

Viene poi definita la funzione per l'**aggiornamento** della **tabella di routing**:

```
def updateTable(self, neighbors):
    """
    Updates the routing table of the current node based on information from its neighbors.

    Args:
        neighbors : dict
        A dictionary of neighboring nodes and the weights of the links to them.
        Keys are Node objects (neighbors), and values are the weights (int).
    """
    for neighbor, weight in neighbors.items():
        for node, infos in neighbor.routingTable.items():
            if node not in self.routingTable:
                self.routingTable.update({node: (weight+infos[0], neighbor.name)})
            elif weight+infos[0] < self.routingTable.get(node)[0]:
                self.routingTable.update({node: (weight+infos[0], neighbor.name)})
```

Questa prende come parametro i vicini del nodo sottoforma di dizionario per il quale la chiave è rappresentata dall'oggetto *Node* del vicino e il valore è il peso del collegamento tra i due. Controlla quindi le routing table che gli vengono passate da ogni vicino. Se il nodo presente nella routing table non è presente nella routing table del mio nodo, allora lo inserisce settando opportunamente la distanza e il next hop. Questo avviene anche nel caso in cui il mio nodo si renda conto che passare per il vicino per arrivare ad un nodo che era già presente nella routing table gli conviene, ovvero il costo complessivo è inferiore.

Si implementa inoltre una funzione per stampare la routing table del nodo con tutte le informazioni:

```
def printRoutingTable(self):
    """
    Print the routing table of the node.
    Print for each reachable node distance and next hop name.
    """
    for node, (distance, hop) in self.routingTable.items():
        print("Node - " + node + " Distance - " + str(distance) + " Next Hop - " + hop)
```

Nel **main** viene poi creata la rete con i nodi e vengono create le liste dei vicini:

```
# Nodes of the network.
nodes = {
    "A" : Node("A"),
    "B" : Node("B"),
    "C" : Node("C"),
    "D" : Node("D"),
    "E" : Node("E"),
    "F" : Node("F"),
    "G" : Node("G")
}

# Neighbors for each node and weight of each link.
network = {
    "A" : {nodes.get("B"): 2, nodes.get("C"):5},
    "B" : {nodes.get("C"): 2, nodes.get("A"):2 },
    "C" : {nodes.get("A"): 5, nodes.get("B"):2, nodes.get("D"):4, nodes.get("E"):6 },
    "D" : {nodes.get("C"): 4, nodes.get("B"):8, nodes.get("F"):4, nodes.get("E"):3 },
    "E" : {nodes.get("C"): 6, nodes.get("D"):3, nodes.get("G"):8},
    "F" : {nodes.get("D"): 4, nodes.get("G"):5 },
    "G" : {nodes.get("E"): 8, nodes.get("F"):5 },
}
```

Viene infine applicato l'algoritmo che **aggiorna** le routing table dei diversi nodi per un numero di volte pari al numero di nodi presenti nella rete, al fine di garantire convergenza:

```
# Iterate the exchange of routing table between nodes for number of nodes time to
# reach convergence.
for iteration in range(len(nodes)):
    for name, node in nodes.items():
        print("Routing table of node " + name + " at iteration n." + str(iteration))
        node.updateTable(network.get(node.name))
        node.printRoutingTable()
```

Ad ogni iterazione viene stampata la routing table del nodo al fine di visualizzare come cambiano queste tabelle nel corso del tempo.