Lab Project 7: **Link State Routing**

- **Number of members per group: 1 or 2**
- **Use Python**

In this lab we implement Dijkstra link state routing using Python. We add nodes and edge to the graph object and then compute the shortest path between a source node and other nodes. In addition to printing the shortest paths, the whole graph and shortest path graph are presented graphically.

The following pseudo-code shows Dijkstra algorithm:

```
Initialization:
N' = {u}
for all nodes v
      if v is a neighbor of u then
            D(v) = c(u,v)
      else
            D(v) = ∞
Loop
find w not in N' such that D(w) is a minimum
add w to N'
update D(v) for each neighbor v of w and not in N':
      D(v) = min( D(v), D(w) + c(w,v) )
      /* new cost to v is either old cost to v or known least path cost to w
      plus cost from w to v */
until N'= N
```

The program includes two important classes: **Vertex** and **Graph**.

The attributes of the Vertex class are as follows:

```
self.id = node
self.adjacent = {}
self.distance = sys.maxsize
self.visited = False
self.previous = None
```

The **id** variable is the identification letter/number for this node. For example, we use identifications such as ‘w’, ‘u’, and ‘v’ to label the nodes.
The **adjacent** dictionary holds the neighbors and link cost to each neighbor. For example, an entry ‘u’: 6 means this node is connected to node ‘u’ through link cost 6.
The **distance** variable (which corresponds to D(v) in the algorithm shown above) is initially set to infinity, and then later during the execution of the Dijkstra algorithm it is updated based on the cost towards the source node.
The **visited** variable is marked as True when the Dijkstra algorithm visits this node and moves it into the list N’.
The **previous** variable represents the previous node on the shortest path from this node to the source node.

The Vertex class includes methods to set and get these attributes. For example, the following function returns the id of the neighbors:

```
    def get_connections(self):
        return self.adjacent.keys()
```

The next important class is **Graph**. The attributes of this class are as follows:

```
self.vert_dict = {}
self.num_vertices = 0
```

The **vert_dict** variable represents the nodes of the graph. For example, an entry 'u': vertex_u represents node id 'u' and the Vertex object presenting this node. Please note that the key is node id and the value is Vertex object. This provides a simple way to map a node id to its actual Vertex object.
The **num_vertices** variable shows the total number of vertices in the network.

An important method of the Graph class is:

```
    def add_vertex(self, node):
        self.num_vertices = self.num_vertices + 1
        new_vertex = Vertex(node)
        self.vert_dict[node] = new_vertex
        return new_vertex
```

This method adds a Vertex to the Graph. The line `new_vertex = Vertex(node)` creates a new Vertex object with the id '`node`' passed to the function. This object is then added as the value of key value 'node'.

The following method adds an edge to the graph:

```
    def add_edge(self, frm, to, cost = 0):

        if frm not in self.vert_dict:
            self.add_vertex(frm)
        if to not in self.vert_dict:
            self.add_vertex(to)

        self.vert_dict[frm].add_neighbor(self.vert_dict[to], cost)
        self.vert_dict[to].add_neighbor(self.vert_dict[frm], cost)

        self.netx.add_edge(frm, to, weight=cost)
```

Please note that this method first checks if the two nodes at the two ends of this link (denoted as '`to`' and '`frm`') are in the graph. If not, the method first adds these nodes to the graph. Then, the neighborhood dictionaries of the two nodes are updated, as follows: Recall that the value corresponding to a key in dictionary `vert_dict` represents the actual vertex object corresponding to that node id. So, the value corresponding to `vert_dict[frm]` is the Vertex object representing node id '`frm`'. We then call the method `add_neighbor` to add Vertex '`to`' as a neighbor of this node.

The actual computation of the shortest path between a node and other nodes in the graph is performed by the following function:

```
    def dijkstra(aGraph, start)
```

The first parameter of this function is a Graph object, and the second parameter is the source node.

Instead of linear search of the nodes in N', in each iteration we use a min-heap to find the node with minimum cost to the 'start' node.

The list of unvisited nodes is represented by `unvisited_queue`, which is initially prepared as follows:

```
unvisited_queue = [(v.get_distance(), id(v), v) for v in aGraph]
```

To heapify this list we can call the following function:

```
heapq.heapify(unvisited_queue)
```

For graphical presentation of the network we use the following module:

```
import networkx as nx
```

We create an instance of the nx.Graph() object, called `netx`, inside the Graph class as follows:

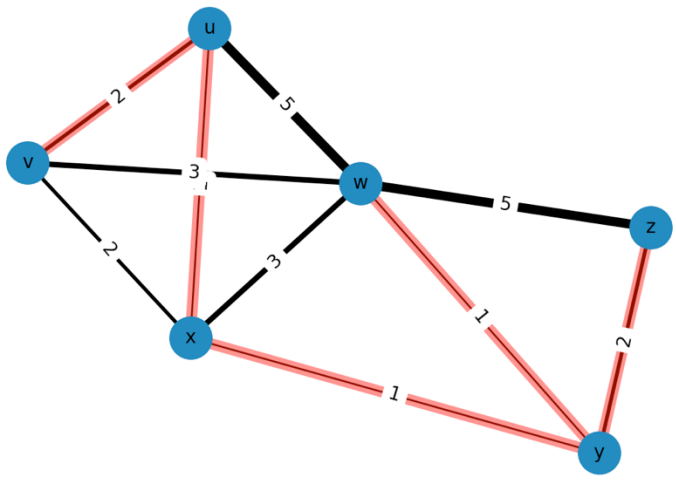
```
self.netx = nx.Graph()
```

Then, whenever an edge/node is added to the Graph object, it should also be added to the **`netx`** object as well. For example, in the `add_edge` method, we add the following line to add an edge:

```
self.netx.add_edge(frm, to, weight=cost)
```

The comments provided in the code provides you with enough information to complete this project.

The graphical output of the program must look like the following picture (ignore node layout):



Deliverables:

- **Demo your project to the TA in the lab**
- **Submit your code to Camino**