CS-102: Data Structures & Algorithms
Lab 09 – Graphs

Habib University

Objectives: In this lab students will build a library of elementary methods to create and manipulate graph data structure. These methods will be used in subsequent labs to implement different graph algorithms.

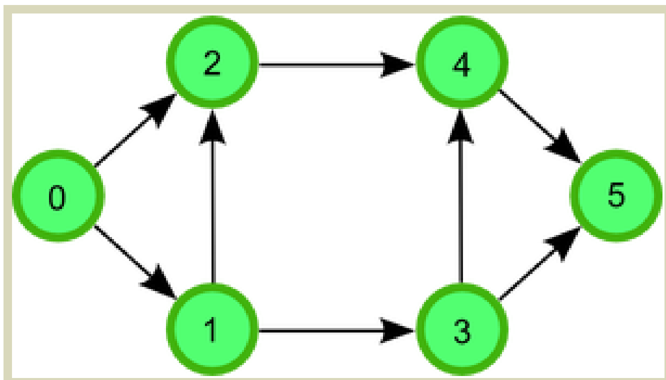


Figure 1.

Python Implementation:

1. **Adjacency matrix: Using a list of lists**
   Adjacency matrix is a widely used graph data structure used to store graph. We can use list of lists for this purpose.

```
adjmatrix = [
            [0, 1, 1, 0, 0, 0],
            [0, 0, 1, 1, 0, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 1],
            [0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0]
            ]
```

2. **Adjacency list: Using a dictionary**

   We can use a dictionary which contains the pair (key, record) = (node, adjList). This is in particular useful if the nodes are not integers but for example strings.

```
Graph = {
            0: [1, 2],
            1: [2, 3],
            2: [4],
            3: [4, 5],
            4: [5],
            5: []
        }
```

   Here, 0: [1, 2] means vertex 0 has the neighbors 1, 2. Similarly, 5: [] means vertex 5 has no neighbors.

Helper Functions

We will use adjacency list to create graph G.

Create following helper functions which would be used in exercises given below.

addNodes(G, nodes): This function will take a graph G and a list of nodes as parameters. It will add the nodes in the list in G and return the Updated Graph.

```
G = {}
nodes = [0, 1, 2, 3, 4, 5]
G = addNodes(G, nodes)
```

```
G = {
0: [],
1: [],
2: [],
3: [],
4: [],
5: []
}
```

addEdges(G, edges, directed = False): This function will take a graph G and a list of edges E as parameters. It will add the edges in the graph G and return the Updated Graph.

<u>Format of list of edges:</u>

edges = [(u1 , v1 , w1) , (u2 , v2 , w2),…]

Here, u = start node, v = end node, w = weight (for un-weighted graph default value of w is 1.)

```
G = {0: [], 1: [], 2: [], 3: [], 4: [], 5: []
}

edge_list = [(0, 1, 1), (0, 2, 1), (1, 2, 1),
 (1, 3, 1), (2, 4, 1), (3, 4, 1), (3, 5, 1),
(4, 5, 1)]

G = addEdges(G, edges, True)
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}
```

```
G = {0: [], 1: [], 2: [], 3: [], 4: [], 5: []
}

edge_list = [(0, 1, 1), (0, 2, 1), (1, 2, 1),
 (1, 3, 1), (2, 4, 1), (3, 4, 1), (3, 5, 1),
(4, 5, 1)]

G = addEdges(G, edges, False)
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}
```

```
G = {0: [], 1: [], 2: [], 3: [], 4: [], 5: []
}

edge_list = [(0, 1, 21), (0, 2, 15), (1, 2, 1
0), (1, 3, 70), (2, 4, 50), (3, 4, 24), (3, 5
, 39), (4, 5, 99)]

G = addEdges(G, edges, True)
```

```
G = {
0: [(1, 21), (2, 15)],
1: [(2, 10), (3, 70)],
2: [(4, 50)],
3: [(4, 24), (5, 39)],
4: [(5, 99)],
5: []
}
```

```
G = {0: [], 1: [], 2: [], 3: [], 4: [], 5: []
}

edge_list = [(0, 1, 21), (0, 2, 15), (1, 2, 1
0), (1, 3, 70), (2, 4, 50), (3, 4, 24), (3, 5
, 39), (4, 5, 99)]

G = addEdges(G, edges, False)
```

```
G = {
0: [(1, 21), (2, 15)],
1: [(0, 21), (2, 10), (3, 70)],
2: [(0, 15), (1, 10), (4, 50)],
3: [(1, 70), (4, 24), (5, 39)],
4: [(3, 24), (2, 50), (5, 99)],
5: [(3, 39), (4, 99)]
}
```

listOfNodes(G): This function will take a graph G as a parameter and return a List of the Nodes in the graph G.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

print(listOfNodes(G))
```

```
[0, 1, 2, 3, 4, 5]
```

listOfEdges(G, directed = False): This function will take a graph G as a parameter and return a List of the Edges in the graph G.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
 }

print(listOfEdges(G, True))
```

```
[(0, 1, 1), (0, 2, 1), (1, 2, 1), (1,
3, 1), (2, 4, 1), (3, 4, 1),
(3, 5, 1), (4, 5, 1)]
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}

print(listOfEdges(G, False))
```

```
[(0, 1, 1), (0, 2, 1), (1, 2, 1), (1,
3, 1), (2, 4, 1), (3, 4, 1),
(3, 5, 1), (4, 5, 1)]
```

printIn_OutDegree(G): This function will take a directed graph G as a parameter and print In Degree and Out Degree for each node in the Graph.

(http://mathonline.wikidot.com/out-degree-sequence-and-in-degree-sequence)

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
 }

print(printIn_OutDegree(G))
```

```
0 => In-Degree: 0, Out-Degree: 2
1 => In-Degree: 1, Out-Degree: 2
2 => In-Degree: 2, Out-Degree: 1
3 => In-Degree: 1, Out-Degree: 2
4 => In-Degree: 2, Out-Degree: 1
5 => In-Degree: 2, Out-Degree: 0
```

printDegree(G): This function will take an undirected graph G as a parameter and print the degree for each node in the Graph.

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}

print(printDegree(G))
```

```
0 => 2
1 => 3
2 => 3
3 => 3
4 => 3
5 => 2
```

getNeighbors(G, node): The function will take an undirected graph G and a node as a parameter and will return the list of its neighboring nodes.

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}

print(getNeighbors(G, 0))
```

```
[1, 2]
```

getInNeighbors(G, node): The function will take a directed graph G and a node as a parameter and will return its in-neighboring nodes.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
 }

print(getInNeighbours(G, 0))
```

```
[]
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}


print(getInNeighbours(G, 2))
```

```
[0, 1]
```

getOutNeighbors(G, node): The function will take a directed graph G and a node as a parameter and will return its out-neighboring nodes.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
 }
print(getOutNeighbours(G, 0))
```

```
[1, 2]
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

print(getOutNeighbours(G, 5))
```

```
[]
```

getNearestNeighbor(G, node): The function will take a weighted undirected graph G and a node and return its nearest neighboring nodes.

```
G = {
0: [(1, 21), (2, 15)],
1: [(0, 21), (2, 10), (3, 70)],
2: [(0, 15), (1, 10), (4, 50)],
3: [(1, 70), (4, 24), (5, 39)],
4: [(3, 24), (2, 50), (5, 99)],
5: [(3, 39), (4, 99)]
}

print(getNearestNeighbor(G, 0))
```

```
2
```

isNeighbor(G, Node1, Node2): The function will take a directed graph G , Node1 and Node2 as a parameter and return True if Node 2 is a neighbor of Node 1 , False otherwise.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

print(isNeighbor(G, 0, 1))
```

```
True
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

print(isNeighbor(G, 1, 0))
```

```
False
```

removeNode(G, node): This function will take a graph G and a node as a parameter. It will remove that node and all edges of that node and return the Updated Graph. In an un-directed graph, you may need to update other values in dictionary to remove a particular node.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

G = removeNode(G, 1)
```

```
G = {
0: [(2, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}

G = removeNode(G, 1)
```

```
G = {
0: [(2, 1)],
2: [(0, 1), (4, 1)],
3: [(4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}
```

removeNodes(G, nodes): This function will take a graph G and a list of nodes as parameters. It will remove the nodes in the list in G and return the Updated Graph.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

G = removeNodes(G, [1, 2])
```

```
G = {
0: [],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}

G = removeNode(G, [1, 2])
```

```
G = {
0: [],
3: [(4, 1), (5, 1)],
4: [(3, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}
```

displayGraph(G): This function will take a graph G as a parameter and print Adjacency list representation of the graph G.

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}

DisplayGraph(G)
```

```
G = {
0: [(1, 1), (2, 1)],
1: [(0, 1), (2, 1), (3, 1)],
2: [(0, 1), (1, 1), (4, 1)],
3: [(1, 1), (4, 1), (5, 1)],
4: [(3, 1), (2, 1), (5, 1)],
5: [(3, 1), (4, 1)]
}
```

display_adj_matrix(G): This function will take a graph G (adjacency list) as a parameter and print the adjacency matrix representation of the graph G.

```
G = {
0: [(1, 1), (2, 1)],
1: [(2, 1), (3, 1)],
2: [(4, 1)],
3: [(4, 1), (5, 1)],
4: [(5, 1)],
5: []
}

display_adj_matrix(G)
```

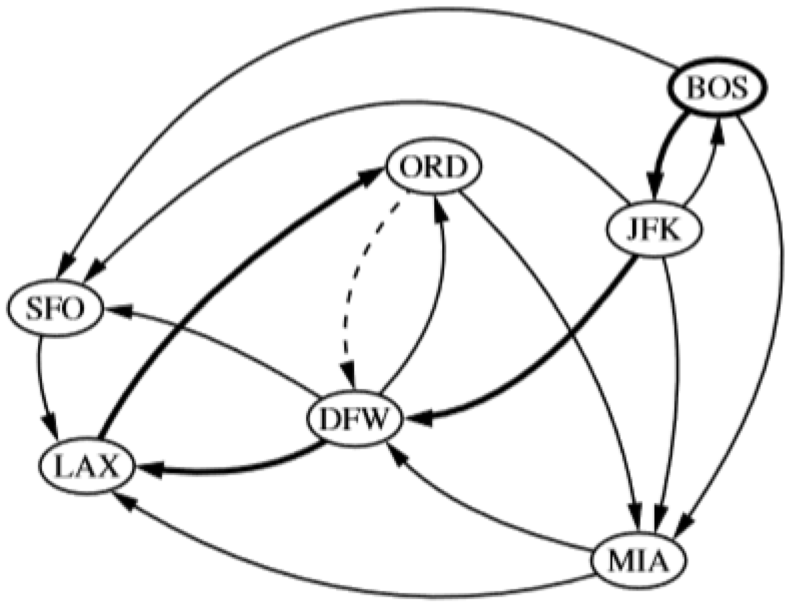
```
[[0, 1, 1, 0, 0, 0],
[0, 0, 1, 1, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 1, 1],
[0, 0, 0, 0, 0, 1],
[0, 0, 0, 0, 0, 0]]
```

```
G = {
0: [(1, 21), (2, 15)],
1: [(2, 10), (3, 70)],
2: [(4, 50)],
3: [(4, 24), (5, 39)],
4: [(5, 99)],
5: []
}

display_adj_matrix(G)
```

```
[[0, 21, 15, 0, 0, 0],
[0, 0, 10, 70, 0, 0],
[0, 0, 0, 0, 50, 0],
[0, 0, 0, 0, 24, 39],
[0, 0, 0, 0, 0, 99],
[0, 0, 0, 0, 0, 0]]
```

Test Cases for Sample Directed Graph:




```
>>> nodes = ['BOS', 'ORD', 'JFK', 'DFW', 'MIA', 'SFO', 'LAX']

>>> edges = [('BOS', 'JFK', 1) , ('BOS', 'MIA', 1), ('BOS', 'SFO', 1), ('JFK', 'B
OS', 1),
('JFK', 'SFO', 1), ('JFK', 'MIA', 1), ('JFK', 'DFW', 1), ('ORD', 'MIA', 1),
('ORD', 'DFW', 1), ('MIA', 'DFW', 1), ('MIA', 'LAX', 1), ('DFW', 'ORD', 1),
('DFW', 'SFO', 1), ('DFW', 'LAX', 1), ('SFO', 'LAX', 1), ('LAX', 'ORD', 1)]

>>> G = {}

>>> print(addNodes(G,nodes))
 {'BOS': [], 'ORD': [], 'JFK': [], 'DFW': [], 'MIA': [], 'SFO': [], 'LAX': []}
>>> print(addEdges(G,edges,True))
 {'BOS': [('JFK', 1), ('MIA', 1), ('SFO', 1)], 'ORD': [('MIA', 1), ('DFW', 1)], '
JFK': [('BOS', 1), ('SFO', 1), ('MIA', 1), ('DFW', 1)], 'DFW': [('ORD', 1), ('SFO
', 1), ('LAX', 1)], 'MIA': [('DFW', 1), ('LAX', 1)], 'SFO': [('LAX', 1)], 'LAX':
[('ORD', 1)]}

>>> print(listOfNodes(G))
['BOS', 'ORD', 'JFK', 'DFW', 'MIA', 'SFO', 'LAX']
>>> print(listOfEdges(G))
 [('BOS', 'JFK', 1), ('BOS', 'MIA', 1), ('BOS', 'SFO', 1), ('ORD', 'MIA', 1), ('O
RD', 'DFW', 1), ('JFK', 'SFO', 1), ('JFK', 'MIA', 1), ('JFK', 'DFW', 1), ('DFW',
'SFO', 1), ('DFW', 'LAX', 1), ('MIA', 'DFW', 1), ('MIA', 'LAX', 1), ('SFO', 'LAX'
, 1), ('LAX', 'ORD', 1)]
```

```
>>> printIn_OutDegree(G)
In Degree
'BOS' = 1, 'ORD' = 2 , 'JFK' = 1 , 'DFW' = 3 , 'MIA' = 3, 'SFO' = 3, 'LAX' = 3
Out Degree
'BOS' = 3, 'ORD' = 2 , 'JFK' = 4 , 'DFW' = 3 , 'MIA' = 2, 'SFO' = 1, 'LAX' = 1

>>> getInNeighbors(G, 'BOS')
['JFK']

>>> getOutNeighbors(G, 'BOS')
['SFO', 'JFK', 'MIA']

>>> isNeighbor(G, 'MIA', 'DFW')
True

>>> isNeighbor(G, 'DFW', 'MIA')
False
```