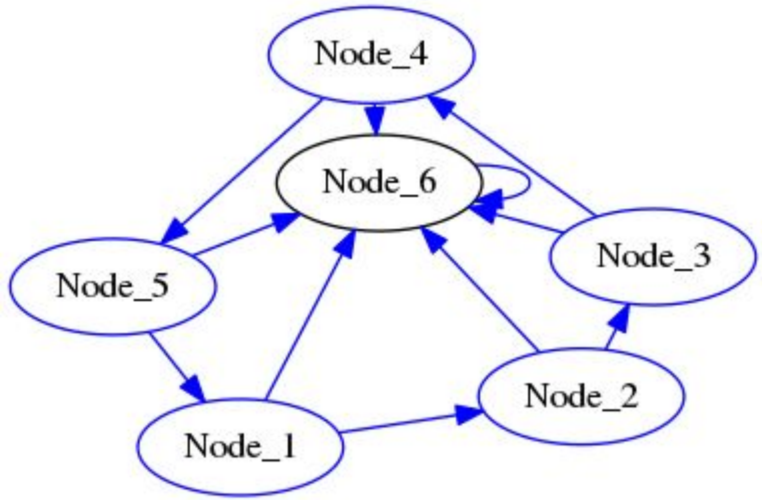
Scientific Programming Practical 19

Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Graphs

Graphs are mathematical structures made of two key elements: **nodes** (or **vertices**) and **edges**. Nodes are things that we want to represent and edges are relationships among the objects.



Mathematically, a graph $G=(N,E)$ where N is a set of nodes and $E=N\times N$ is the set of edges.

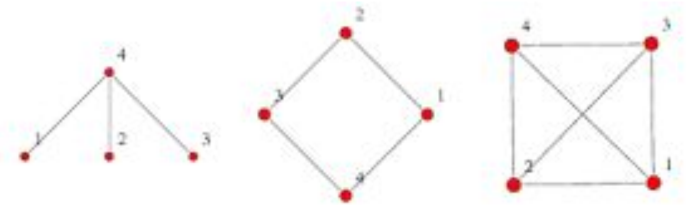
Graph: ADT

Graphs are dynamic data structures in which nodes and edges can be added/removed.

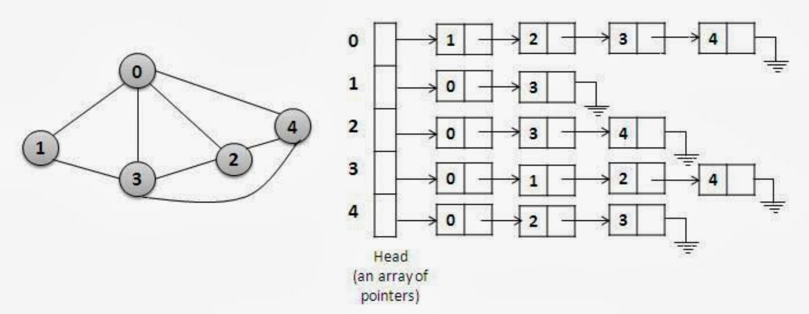
GRAPH	
Graph( )	% Create a new graph
SET size()	% Returns the number of nodes
SET V()	% Returns the set of all nodes
SET adj(NODE u)	% Returns the set of nodes adjacent to u
insertNode(NODE u)	% Add node u to the graph
insertEdge(NODE u, NODE v)	% Add edge (u, v) to the graph
deleteNode(NODE u)	% Removes node u from the graph
deleteEdge(NODE u, NODE v)	% Removes edge (u, v) from the graph

Two possible implementations

Graphs can be implemented as **adjacency matrices** or **adjacency linked lists**.



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$



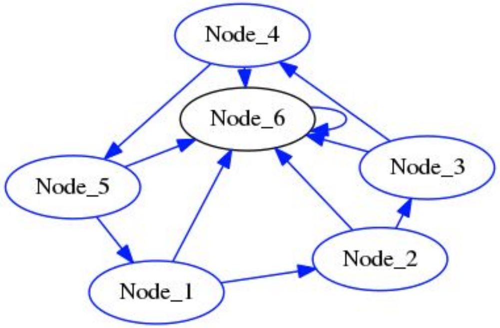
Adjacency List Representation of Graph

Adjacency linked list: implementation

```python
class DiGraphLL:
    def __init__(self):
        """Every node is an element in the dictionary.
        The key is the node id and the value is a dictionary
        with key second node and value the weight
        """
        self.__nodes = dict()

    def insertNode(self, node):
        test = self.__nodes.get(node, None)

        if test == None:
            self.__nodes[node] = {}
            #print("Node {} added".format(node))

    def insertEdge(self, node1, node2, weight):
        test = self.__nodes.get(node1, None)
        test1 = self.__nodes.get(node2, None)
        if test != None and test1 != None:
            #if both nodes exist othewise don't do anything
            test = self.__nodes[node1].get(node2, None)
            if test != None:
                exStr= "Edge {} --> {} already existing.".format(node1,
                                                                  node2)
                raise Exception(exStr)
            else:
                #print("Inserted {}-->{} ({})".format(node1,node2,weight))
                self.__nodes[node1][node2] = weight
```

Instead of a linked list we use a dictionary for speed

{ 'Node_5': {'Node_6': 1, 'Node_1': 0.5},
'Node_1': {'Node_6': 1, 'Node_2': 0.5},
'Node_2': {'Node_6': 1, 'Node_3': 0.5},
'Node_6': {'Node_6': 1},
'Node_3': {'Node_6': 1, 'Node_4': 0.5},
'Node_4': {'Node_6': 1, 'Node_5': 0.5} }

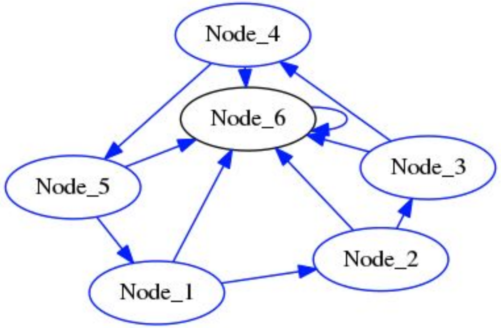


Adjacency linked list: implementation

```python
class DiGraphLL:
    def __init__(self):
        """Every node is an element in the dictionary.
        The key is the node id and the value is a dictionary
        with key second node and value the weight
        """
        self.__nodes = dict()

    def insertNode(self, node):
        test = self.__nodes.get(node, None)

        if test == None:
            self.__nodes[node] = {}
            #print("Node {} added".format(node))

    def insertEdge(self, node1, node2, weight):
        test = self.__nodes.get(node1, None)
        test1 = self.__nodes.get(node2, None)
        if test != None and test1 != None:
            #if both nodes exist othewise don't do anything
            test = self.__nodes[node1].get(node2, None)
            if test != None:
                exStr= "Edge {} --> {} already existing.".format(node1,
                                                                 node2)
                raise Exception(exStr)
            else:
                #print("Inserted {}-->{} ({})".format(node1,node2,weight))
                self.__nodes[node1][node2] = weight
```

```python
def adjacent(self, node):
    """returns a list of nodes connected to node"""
    ret = []
    test = self.__nodes.get(node, None)
    if test != None:
        for n in self.__nodes:
            if n == node:
                #all outgoing edges
                for edge in self.__nodes[node]:
                    ret.append(edge)
            else:
                #all incoming edges
                for edge in self.__nodes[n]:
                    if edge == node:
                        ret.append(n)

    return ret
def adjacentEdge(self, node, incoming = True):
    """
    If incoming == False
    we look at the edges of the node
    else we need to loop through all the nodes.
    An edge is present if there is a
    corresponding entry in the dictionary.
    If no such nodes exist returns None
    """
    ret = []
    if incoming == False:
        otherNode = self.__nodes.get(node,None)
        if otherNode != None:
            for e in otherNode:
                w = self.__nodes[node][e]
                ret.append((node, e, w))
            return ret
    else:
        for n in self.__nodes:
            other = self.__nodes[n].get(node,None)
            if edge != None:
                ret.append((n,node, other))
        return ret
```

Adjacency linked list: implementation




```python
if __name__ == "__main__":
    G = DiGraphLL()
    for i in range(6):
        n = "Node_{}".format(i+1)
        G.insertNode(n)

    for i in range(0,4):
        n = "Node_" + str(i+1)
        six = "Node_6"
        n_plus = "Node_" + str((i+2) % 6)
        G.insertEdge(n, n_plus,0.5)
        G.insertEdge(n, six,1)
    G.insertEdge("Node_5", "Node_1", 0.5)
    G.insertEdge("Node_5", "Node_6", 1)
    G.insertEdge("Node_6", "Node_6", 1)
    print(G)

    G.insertNode("Node_7")
    G.insertEdge("Node_1", "Node_7", -1)
    G.insertEdge("Node_2", "Node_7", -2)
    G.insertEdge("Node_5", "Node_7", -5)
    G.insertEdge("Node_7", "Node_2", -2)
    G.insertEdge("Node_7", "Node_3", -3)

    print("Size is: {}".format(len(G)))
    print("Nodes: {}".format(G.nodes()))
    print("Graph:")
    print(G)
    G.deleteNode("Node_7")
    G.deleteEdge("Node_6", "Node_2")
    #nodes do not exist! Therefore nothing happens!
    G.insertEdge("72", "25",3)
    print(G)
    print("Nodes: {}".format(G.nodes()))
    G.deleteEdge("72","25")
    print("Nodes: {}".format(G.nodes()))
    print(G)
```

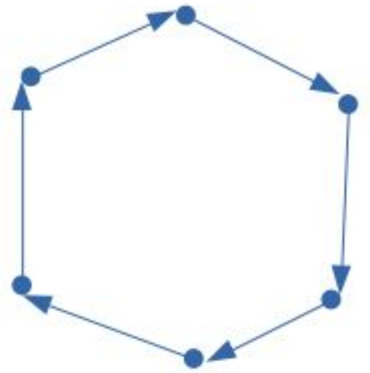
```
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_2 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_6 -- 1 --> Node_6

Size is: 7
Nodes: ['Node_5', 'Node_4', 'Node_7', 'Node_1', 'Node_2', 'Node_3', 'Node_6']
Graph:
Node_5 -- -5 --> Node_7
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_7 -- -2 --> Node_2
Node_7 -- -3 --> Node_3
Node_1 -- -1 --> Node_7
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_2 -- 1 --> Node_6
Node_2 -- -2 --> Node_7
Node_2 -- 0.5 --> Node_3
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_6 -- 1 --> Node_6
```

```
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_2 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_6 -- 1 --> Node_6

Nodes: ['Node_5', 'Node_4', 'Node_3', 'Node_2', 'Node_1', 'Node_6']
Nodes: ['Node_5', 'Node_4', 'Node_3', 'Node_2', 'Node_1', 'Node_6']
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_2 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_6 -- 1 --> Node_6
```

Visits

Traversing a graph means **going through its edges and nodes following the connections that make up the graph**. Graphs can have **cycles** and this makes it quite tricky to visit the graph.

Differently from what seen in the case of trees, we need to keep a structure of **visited** nodes to avoid getting stuck in loops.



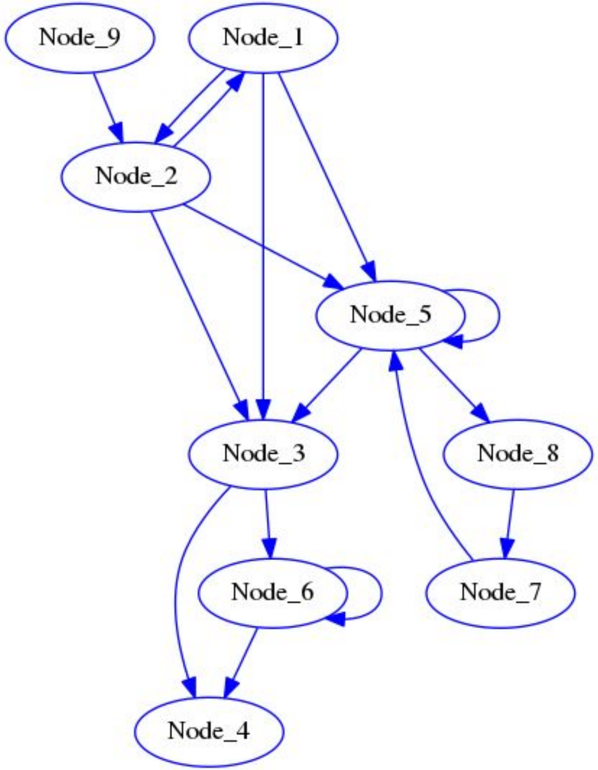
As in the case of Trees, two ways exist to perform a visit of a graph: **depth first search** and **breadth first search**

Depth First Search

Depth First Search visits nodes of the graph going as deep as possible along each path.

This procedure is normally used to travel through **all the nodes** of the graph, and so it might have to be repeated several times (starting each time from a different node) as the output is in general a **forest of DFS trees**.

```
DFS Preorder:

Node_1
        from Node_1 --> Node_5
Node_5
        from Node_5 --> Node_8
Node_8
        from Node_8 --> Node_7
Node_7
        from Node_5 --> Node_3
Node_3
        from Node_3 --> Node_6
Node_6
        from Node_6 --> Node_4
Node_4
        from Node_1 --> Node_2
Node_2
```

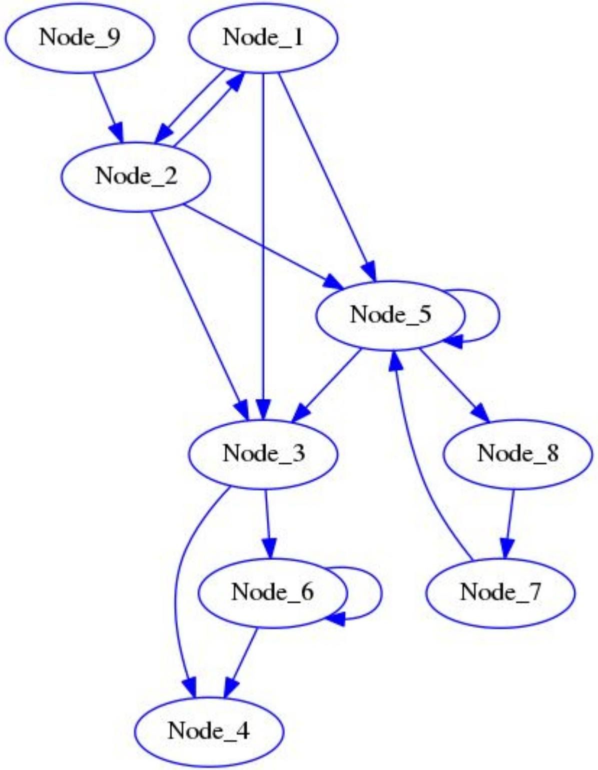


Depth First Search

Depth First Search visits nodes of the graph going as deep as possible along each path.

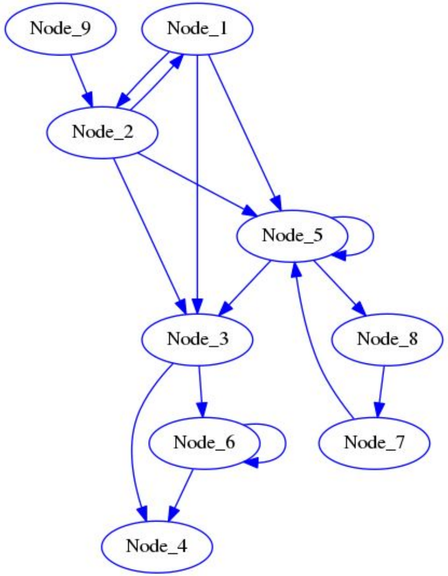
This procedure is normally used to travel through **all the nodes** of the graph, and so it might have to be repeated several times (starting each time from a different node) as the output is in general a **forest of DFS trees**.

```
Postorder:
        from Node_1 --> Node_5
        from Node_5 --> Node_8
        from Node_8 --> Node_7
Node_7
Node_8
        from Node_5 --> Node_3
        from Node_3 --> Node_6
        from Node_6 --> Node_4
Node_4
Node_6
Node_3
Node_5
        from Node_1 --> Node_2
Node_2
Node_1
```



Depth First Search

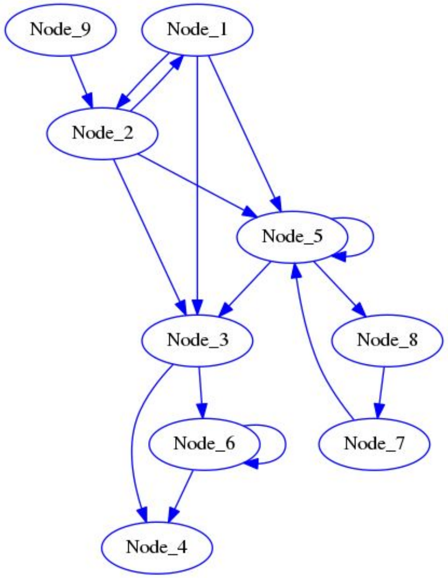
```
import sys
sys.path.append('DiGraphLL')
import DiGraphLL
from collections import deque

class DiGraph(DiGraphLL.DiGraphLL):

    def DFSvisit(self, root, preorder = True, visited = set()):
        visited.add(root)
        if preorder:
            print("{}".format(root))
        #remember that self.adjacentEdge returns:
        #[('node1','node2', weight1), ...('node1', 'nodeX', weightX)]
        outGoingEdges = self.adjacentEdge(root, incoming = False)
        nextNodes = []
        if len(outGoingEdges) > 0:
            nextNodes = [x[1] for x in outGoingEdges]
            #print(nextNodes)
            for nextN in nextNodes:
                if nextN not in visited:
                    print("\tfrom {} --> {}".format(root, nextN))
                    self.DFSvisit(nextN, preorder, visited)
        if not preorder:
            print("{}".format(root))
```



Depth First Search

```python
if __name__ == "__main__":
    G = DiGraph()
    for i in range(1,10):
        G.insertNode("Node_" + str(i))

    G.insertEdge("Node_1", "Node_2",1)
    G.insertEdge("Node_2", "Node_1",1)
    G.insertEdge("Node_1", "Node_3",1)
    G.insertEdge("Node_1", "Node_5",1)
    G.insertEdge("Node_2", "Node_3",1)
    G.insertEdge("Node_2", "Node_5",1)
    G.insertEdge("Node_3", "Node_4",1)
    G.insertEdge("Node_3", "Node_6",1)
    G.insertEdge("Node_5", "Node_3",1)
    G.insertEdge("Node_5", "Node_5",1)
    G.insertEdge("Node_6", "Node_4",1)
    G.insertEdge("Node_6", "Node_6",1)
    G.insertEdge("Node_7", "Node_5",1)
    G.insertEdge("Node_5", "Node_8",1)
    G.insertEdge("Node_8", "Node_7",1)
    G.insertEdge("Node_9", "Node_2",1)

    print("Preorder:")
    G.DFSvisit("Node_1", preorder = True, visited = set())
    print("\nPostorder:")
    G.DFSvisit("Node_1", preorder = False, visited = set())

    print("\nPreorder from Node_9")
    G.DFSvisit("Node_9", preorder = True, visited = set())
```

```
Preorder:
Node_1
        from Node_1 --> Node_5
Node_5
        from Node_5 --> Node_8
Node_8
        from Node_8 --> Node_7
Node_7
        from Node_5 --> Node_3
Node_3
        from Node_3 --> Node_6
Node_6
        from Node_6 --> Node_4
Node_4
        from Node_1 --> Node_2
Node_2

Postorder:
        from Node_1 --> Node_5
        from Node_5 --> Node_8
        from Node_8 --> Node_7
Node_7
Node_8
        from Node_5 --> Node_3
        from Node_3 --> Node_6
        from Node_6 --> Node_4
Node_4
Node_6
Node_3
Node_5
        from Node_1 --> Node_2
Node_2
Node_1
```

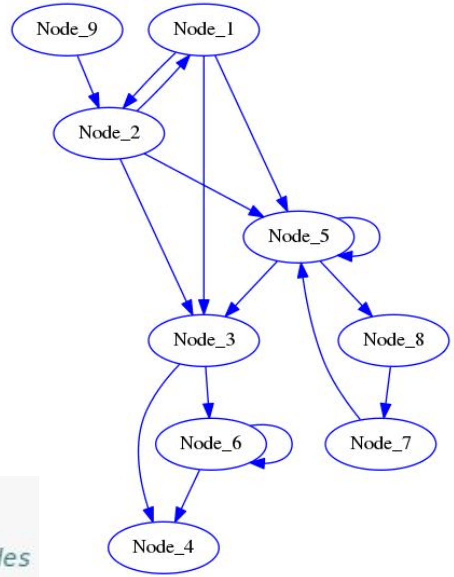



```
Preorder from Node_9
Node_9
        from Node_9 --> Node_2
Node_2
        from Node_2 --> Node_5
Node_5
        from Node_5 --> Node_8
Node_8
        from Node_8 --> Node_7
Node_7
        from Node_5 --> Node_3
Node_3
        from Node_3 --> Node_6
Node_6
        from Node_6 --> Node_4
Node_4
        from Node_2 --> Node_1
Node_1
```

Depth First Search

To make sure all the nodes are visited...

```
def DFS(self, root, preorder = True):
    visited = set()
    #first visit from specified node then check all other nodes
    #set is mutable so the set is going to change!
    print("Starting from {}".format(root))
    self.DFSvisit(root, preorder, visited)

    for node in self.nodes():
        if node not in visited:
            print("Starting from {}".format(node))
            self.DFSvisit(node, preorder, visited)
```

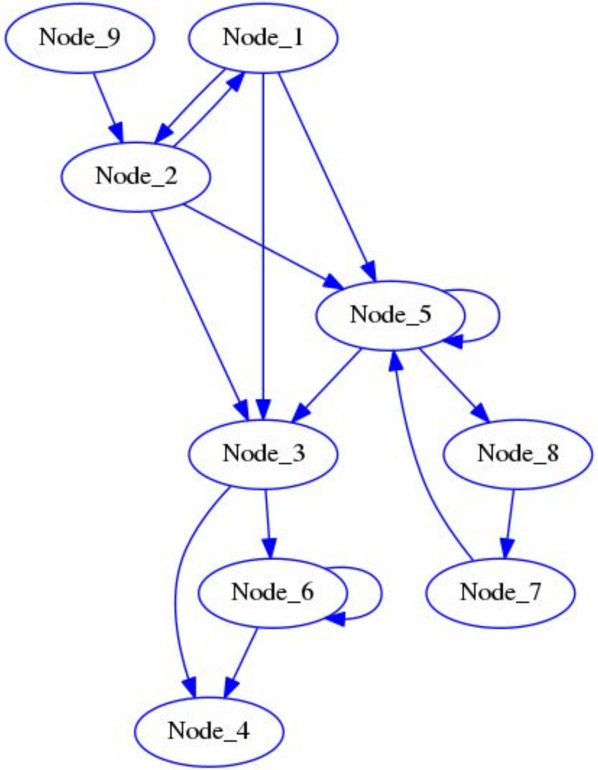


Breadth First Search

Breadth First Search visits all the nodes starting from a *root* node level by level. This means that first **all the nodes at distance 1** from the *root* are visited, then **all the nodes at distance 2** and so on.

This algorithm, in general, does not visit all the nodes, but only those ones that are reachable from the specified root. If all nodes must be visited, another visit should start from a node not touched in the first visit.

```
BFS(Node_1):
From Node_1:
          --> Node_5
          --> Node_2
          --> Node_3
From Node_5:
          --> Node_8
From Node_2:
From Node_3:
          --> Node_6
          --> Node_4
From Node_8:
          --> Node_7
From Node_6:
From Node_4:
From Node_7:
```

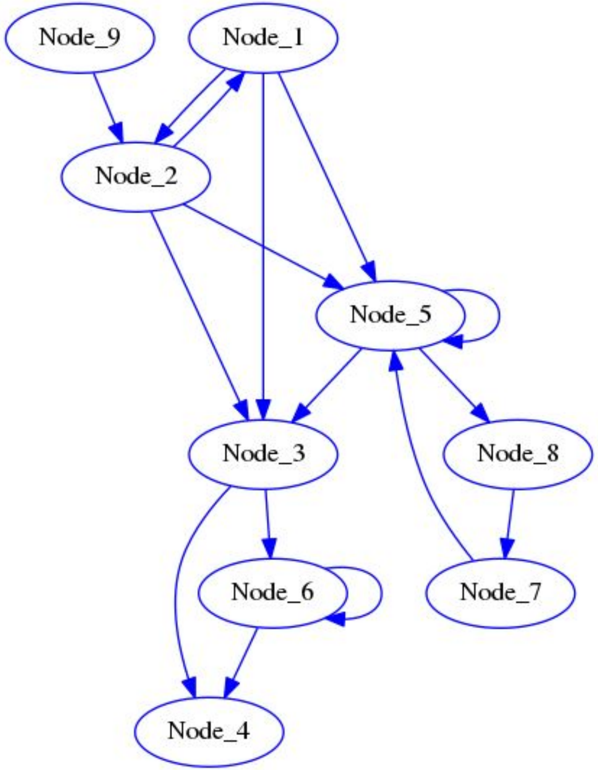


Breadth First Search

Breadth First Search visits all the nodes starting from a *root* node level by level. This means that first **all the nodes at distance 1** from the *root* are visited, then **all the nodes at distance 2** and so on.

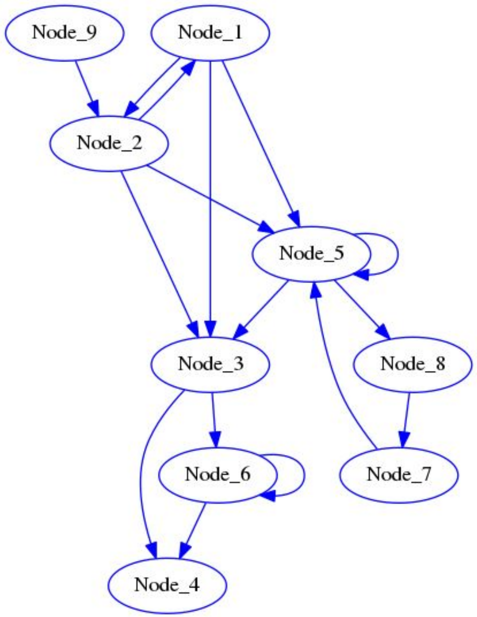
This algorithm, in general, does not visit all the nodes, but only those ones that are reachable from the specified root. If all nodes must be visited, another visit should start from a node not touched in the first visit.

```
BFS(Node_9):
From Node_9:
          --> Node_2
From Node_2:
          --> Node_5
          --> Node_1
          --> Node_3
From Node_5:
          --> Node_8
From Node_1:
From Node_3:
          --> Node_6
          --> Node_4
From Node_8:
          --> Node_7
From Node_6:
From Node_4:
From Node_7:
```



Breadth First Search

```python
import sys
sys.path.append('DiGraphLL')
import DiGraphLL
from collections import deque

class DiGraph(DiGraphLL.DiGraphLL):
    """
    Every node is an element in the dictionary.
    The key is the node id and the value is a dictionary
    with key second node and value the weight
    """
    def BFSvisit(self, root):
        if root in self.graph():
            Q = deque()
            Q.append(root)
            #visited is a set of visited nodes
            visited = set()
            visited.add(root)
            while len(Q) > 0:
                curNode = Q.popleft()
                outGoingEdges = self.adjacentEdge(curNode, incoming = False)
                nextNodes = []
                if outGoingEdges != None:
                    #remember that self.adjacentEdge returns:
                    #[('node1','node2', weight1), ...('node1', 'nodeX', weightX)]
                    nextNodes = [x[1] for x in outGoingEdges]
                print("From {}:".format(curNode))
                for nextNode in nextNodes:
                    if nextNode not in visited:
                        Q.append(nextNode)
                        visited.add(nextNode)
                        print("\t --> {}".format(nextNode ))
```



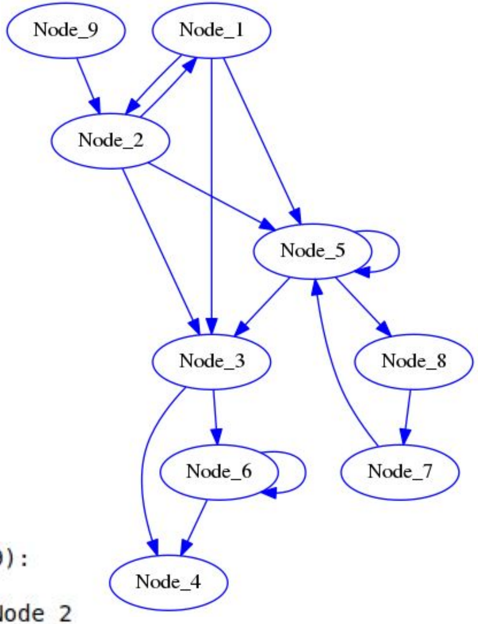
Breadth First Search

```python
if __name__ == "__main__":
    G = DiGraph()
    for i in range(1,10):
        G.insertNode("Node_" + str(i))

    G.insertEdge("Node_1", "Node_2",1)
    G.insertEdge("Node_2", "Node_1",1)
    G.insertEdge("Node_1", "Node_3",1)
    G.insertEdge("Node_1", "Node_5",1)
    G.insertEdge("Node_2", "Node_3",1)
    G.insertEdge("Node_2", "Node_5",1)
    G.insertEdge("Node_3", "Node_4",1)
    G.insertEdge("Node_3", "Node_6",1)
    G.insertEdge("Node_5", "Node_3",1)
    G.insertEdge("Node_5", "Node_5",1)
    G.insertEdge("Node_6", "Node_4",1)
    G.insertEdge("Node_6", "Node_6",1)
    G.insertEdge("Node_7", "Node_5",1)
    G.insertEdge("Node_5", "Node_8",1)
    G.insertEdge("Node_8", "Node_7",1)
    G.insertEdge("Node_9", "Node_2",1)

    print("BFS(Node_1):")
    G.BFSvisit("Node_1")
    print("\nNow BFS(Node_5):")
    G.BFSvisit("Node_5")
    print("\nNow BFS(Node_9):")
    G.BFSvisit("Node_9")
```

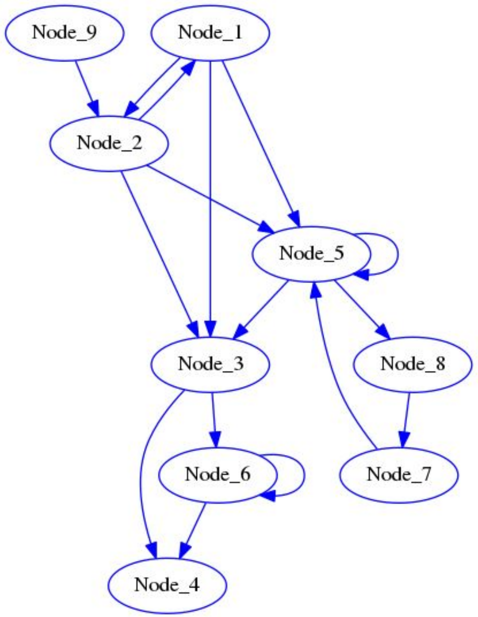
```
BFS(Node_1):
From Node_1:
                --> Node_5
                --> Node_2
                --> Node_3
From Node_5:
                --> Node_8
From Node_2:
From Node_3:
                --> Node_6
                --> Node_4
From Node_8:
                --> Node_7
From Node_6:
From Node_4:
From Node_7:
```

```
Now BFS(Node_5):
From Node_5:
                --> Node_8
                --> Node_3
From Node_8:
                --> Node_7
From Node_3:
                --> Node_6
                --> Node_4
From Node_7:
From Node_6:
From Node_4:
```

```
Now BFS(Node_9):
From Node_9:
                --> Node_2
From Node_2:
                --> Node_5
                --> Node_1
                --> Node_3
From Node_5:
                --> Node_8
From Node_1:
From Node_3:
                --> Node_6
                --> Node_4
From Node_8:
                --> Node_7
From Node_6:
From Node_4:
From Node_7:
```



Breadth First Search



To make sure all the nodes are visited...

```
def BFSvisit(self, root, visited = set()):
    if root in self.graph():
    ...
```

```
def BFS(self, root):
    print("Starting from {}".format(root))
    visited = set()
    visited.add(root)
    self.BFSvisit(root,visited)
    for node in self.nodes():
        if node not in visited:
            print("Starting from {}".format(node))
            self.BFSvisit(node,visited)
```

http://qcbprolab.readthedocs.io/en/latest/practical19.html

Exercises

1. Write a method `shortestPath(self, node1, node2)` that finds the shortest path between two nodes `node1` and `node2` (if it exists) ignoring any weight on the edges.

Hint: modify the BFS in such a way at it stops when the second node is reached (or when no more nodes can be explored). Apply this code twice, that is once from node1->node2 and the other from node2->node1 to find the shortest of the two. You need to change the code in order to obtain the path.

Structuring the code (hints):

a. Implement a method `shortestPath(self,node1,node2)` that finds and prints the shortest path between node1 and node2 (testing both node1 -> node2 and node2 -> node1 and returning the shortest of the two) calling the other two functions;
b. Implement a method `path = findShortestPath(self,node1,node2)` that returns the path going from node1 to node2 if this exist. **Note:** in my implementation, path is a list starting with the terminal node [node2, nodex, nodexx, node1] and ending with the first node of the sought path. The idea behind the implementation of this method is to store somewhere (like in a dictionary) the parent of each visited node and then produce the list above going backwards from node2 up to node1.
c. Implement a method `printPath(self,path)` that gets a path as specified above and prints it out like:

```
Shortest path between: Node_1 and Node_6
Node_1 --> Node_3
    Node_3 --> Node_6
```