IIT Madras
BSc Degree

Home Week-3 Week-5

PDSA - Week 4

Introduction of Graph

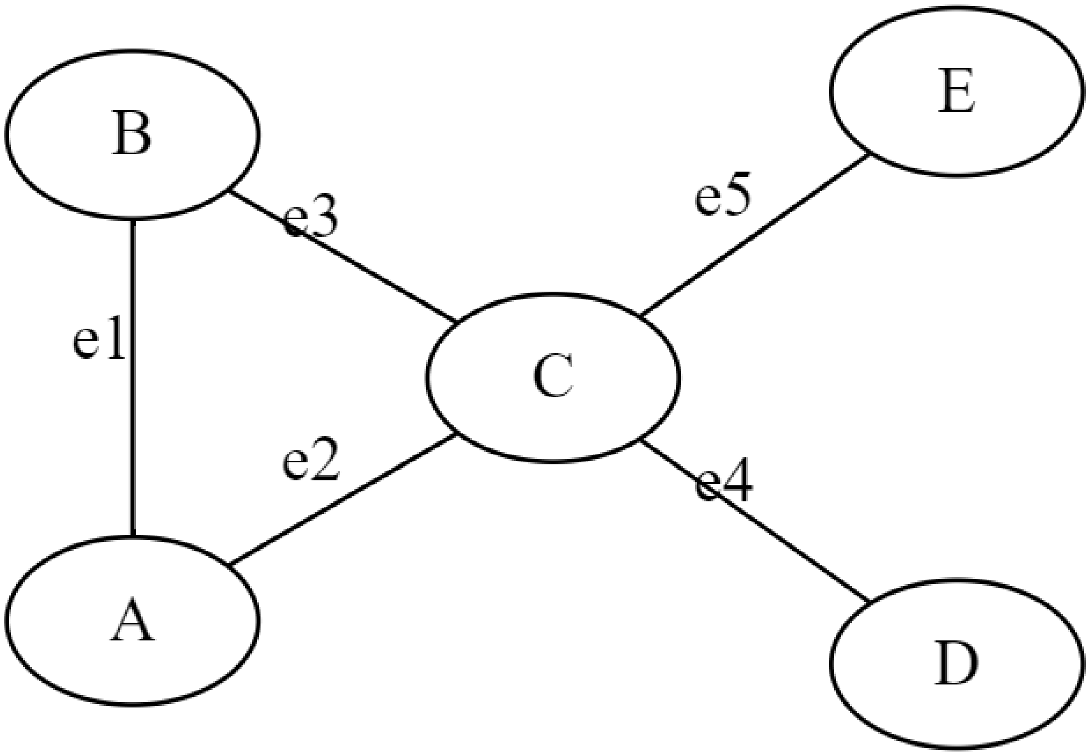
Graph:- It is a non-linear data structure. A graph G consist of a non empty set V where members are called the vertices of graph and the set E where member are called the edges.

G = (V, E)

V = set of vertices

E = set of edges

Example:-

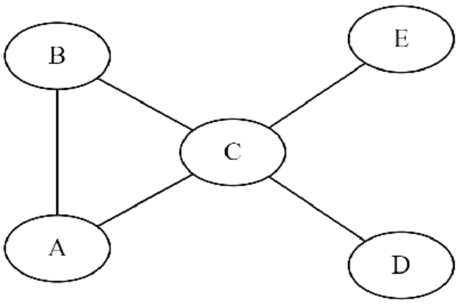


G = (V, E)

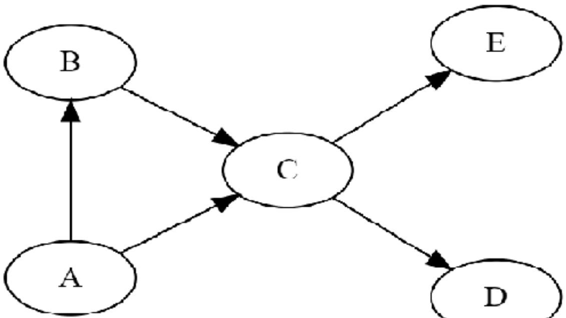
V = {A,B,C,D,E}

E = {e1,e2,e3,e4,e5} or {(A,B), (A,C), (B,C), (C,D), (C,E)}

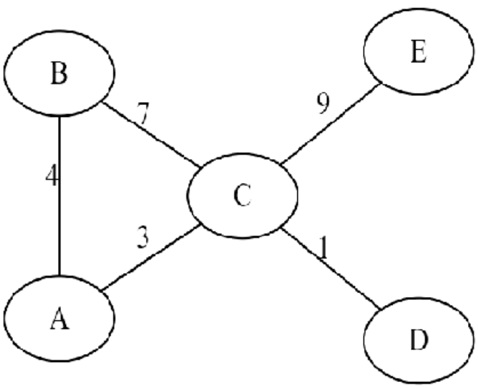
Types of graph



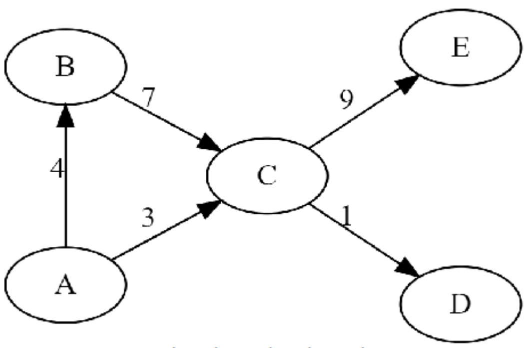
Undirected and unweighted graph



Directed and unweighted graph



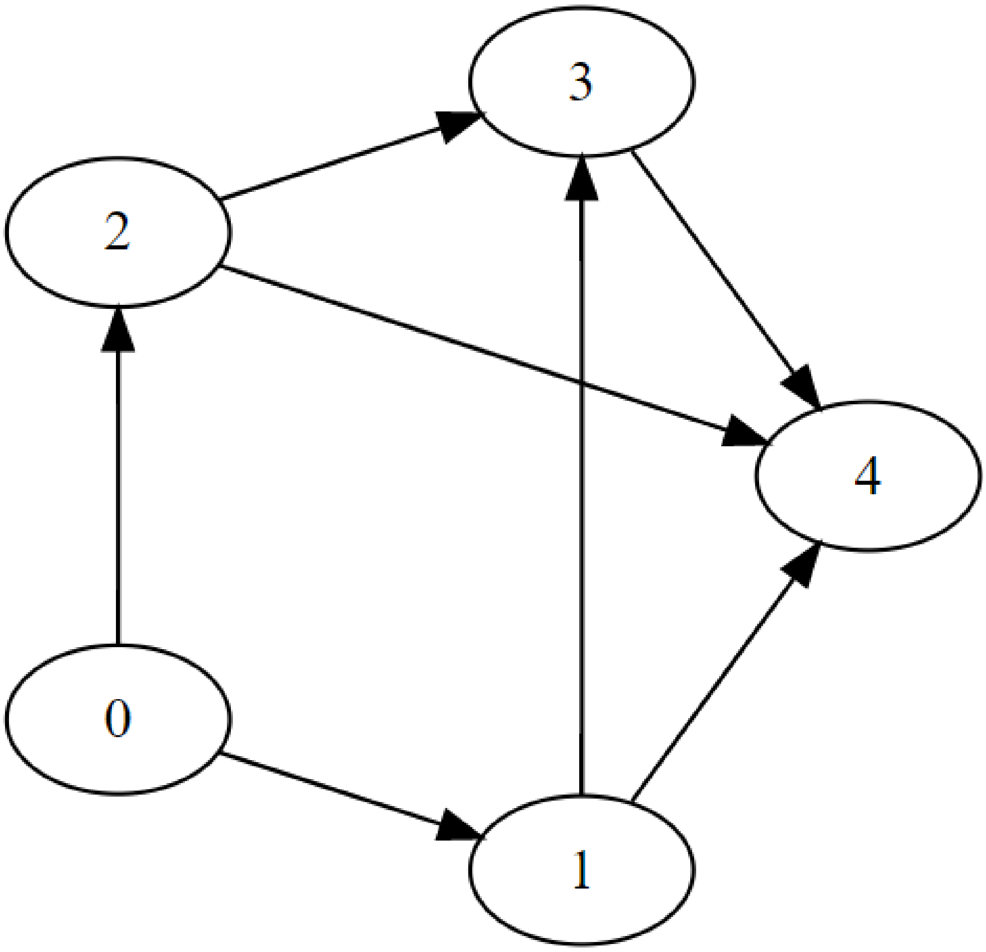
Undirected and weighted graph



Directed and weighted graph

Graph Representation(Unweighted)

For directed graph



G = (V, E)

V = [0,1,2,3,4]

E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]

Adjacency matrix creation(using numpy 2d array) for given directed graph in python

```
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)] # each tuple(u,v) represent edge from u to v
size = len(V)
import numpy as np
AMat = np.zeros(shape=(size,size))
for (i,j) in E:
    AMat[i,j] = 1 # mark 1 if edge present in graph from i to j , otherwise 0
print(AMat)
```

Output adjacency matrix (AMat)

```
[[0. 1. 1. 0. 0.]
 [0. 0. 0. 1. 1.]
 [0. 0. 0. 1. 1.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]]
# AMat[i,j] == 1 represent edge from i to j
```

Adjacency matrix creation(using nested list) for given directed graph in python

```
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
size = len(V)
AMat = []
for i in range(size):
    row = []
    for j in range(size):
        row.append(0)
    AMat.append(row.copy())
for (i,j) in E:
    AMat[i][j] = 1 # mark 1 if edge present in graph from i to j , otherwise 0
print(AMat)
```

Output adjacency matrix (AMat)

```
[[0, 1, 1, 0, 0],
[0, 0, 0, 1, 1],
[0, 0, 0, 1, 1],
[0, 0, 0, 0, 1],
[0, 0, 0, 0, 0]]
# AMat[i][j] == 1 represent edge from i to j
```

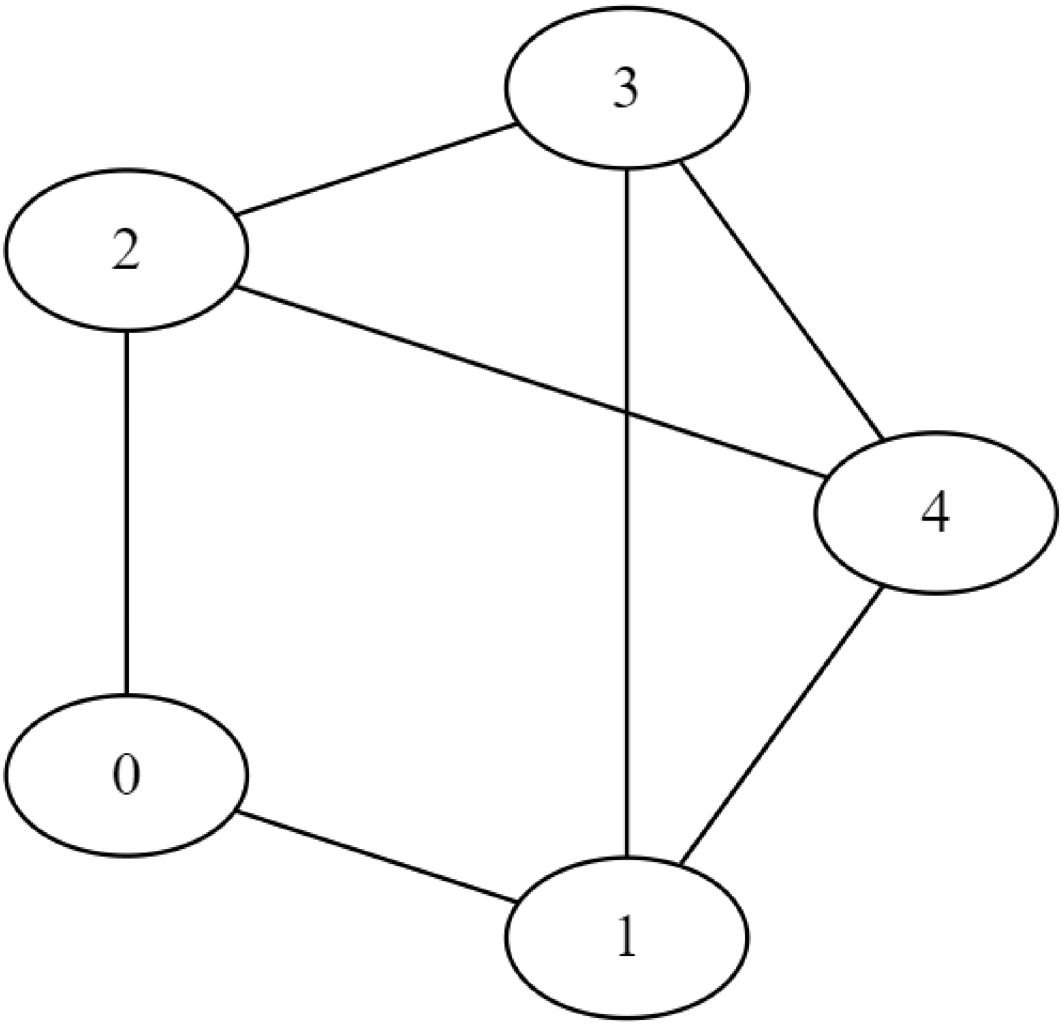
Adjacency list creation(using dictionary) for given directed graph in python

```
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
size = len(V)
AList = {}
# In dictionay AList, for example, AList[i] = [j,k] represent two edge from i to j and i to k
for i in range(size):
    AList[i] = []
for (i,j) in E:
    AList[i].append(j)
print(AList)
```

Output adjacency list (AList)

```
{0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
# for example, AList[i] = [j,k] represent two edge from i to j and i to k
```

For undirected graph



G = (V, E)

V = [0,1,2,3,4]

E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]

Adjacency matrix creation(using numpy 2d array) for given undirected graph in python

```
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
UE = E + [ (j,i) for (i,j) in E] # each edge represented by two tuple (u,v)
and (v,u)
size = len(V)
import numpy as np
AMat = np.zeros(shape=(size,size))
for (i,j) in UE:
    AMat[i,j] = 1 # mark 1 if edge present in graph from i to j , otherwise 0
print(AMat)
```

Output adjacency matrix (AMat)

```
[[0. 1. 1. 0. 0.]
 [1. 0. 0. 1. 1.]
 [1. 0. 0. 1. 1.]
 [0. 1. 1. 0. 1.]
 [0. 1. 1. 1. 0.]]
# AMat[i,j] == 1 represent edge from i to j
```

Adjacency matrix creation(using nested list) for given undirected graph in python

```python
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
UE = E + [ (j,i) for (i,j) in E] # each edge represented by two tuple (u,v) and (v,u)
size = len(V)
AMat = []
for i in range(size):
    row = []
    for j in range(size):
        row.append(0)
    AMat.append(row.copy())
for (i,j) in UE:
    AMat[i][j] = 1 # mark 1 if edge present in graph from i to j , otherwise 0
print(AMat)
```

Output adjacency matrix (AMat)

```
[[0, 1, 1, 0, 0],
[1, 0, 0, 1, 1],
[1, 0, 0, 1, 1],
[0, 1, 1, 0, 1],
[0, 1, 1, 1, 0]]
# AMat[i][j] == 1 represent edge from i to j
```

Adjacency list creation(using dictionary) for given undirected graph in python

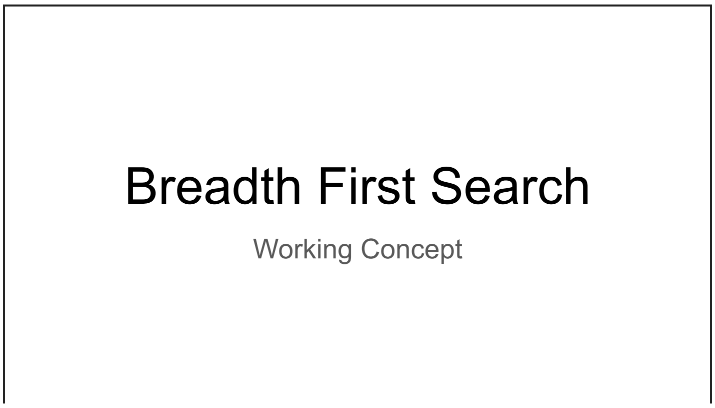
```python
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
UE = E + [ (j,i) for (i,j) in E] # each edge represented by two tuple (u,v) and (v,u)
size = len(V)
AList = {}
# In dictionay AList, for example, AList[i] = [j,k] represent two edge from i to j and i to k
for i in range(size):
    AList[i] = []
for (i,j) in UE:
    AList[i].append(j)
print(AList)
```

Output adjacency list (AList)

```
{0: [1, 2], 1: [3, 4, 0], 2: [4, 3, 0], 3: [4, 1, 2], 4: [1, 2, 3]}
# for example, AList[i] = [j,k] represent two edge from i to j and i to k
```

Graph Traversing Algorithm

Breadth First Search(BFS)



BFS for adjacency list of graph

```python
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
```

```
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))

def BFSList(AList,v):
    visited = {}
    for i in AList.keys():
        visited[i] = False
    q = Queue()

    visited[v] = True
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (not visited[k]):
                visited[k] = True
                q.addq(k)
    return(visited)
AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
print(BFSList(AList,0))
```

Output

```
{0: True, 1: True, 2: True, 3: True, 4: True}
```

BFS for adjacency matrix of graph

```
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))

def neighbours(AMat,i):
    nbrs = []
    (rows,cols) = AMat.shape
```

```python
    for j in range(cols):
        if AMat[i,j] == 1:
            nbrs.append(j)
    return(nbrs)
def BFS(AMat,v):
    (rows,cols) = AMat.shape
    visited = {}
    for i in range(rows):
        visited[i] = False
    q = Queue()

    visited[v] = True
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in neighbours(AMat,j):
            if (not visited[k]):
                visited[k] = True
                q.addq(k)

    return(visited)

V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
size = len(V)
import numpy as np
AMat = np.zeros(shape=(size,size))
for (i,j) in E:
    AMat[i,j] = 1
print(BFS(AMat,0))
```

Output

```
{0: True, 1: True, 2: True, 3: True, 4: True}
```

List of path from vertex `v` using BFS

```python
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))
```

```python

def BFSListPath(AList,v):
    (visited,parent) = ({},{})
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    q = Queue()

    visited[v] = True
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (not visited[k]):
                visited[k] = True
                parent[k] = j
                q.addq(k)

    return(visited,parent)
AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
print(BFSListPath(AList,0))
```

Output

```
({0: True, 1: True, 2: True, 3: True, 4: True}, {0: -1, 1: 0, 2: 0, 3: 1, 4:
1})
```

Find level of vertex `v` using BFS

```python
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))

def BFSListPathLevel(AList,v):
    (level,parent) = ({},{})
    for i in AList.keys():
        level[i] = -1
        parent[i] = -1
    q = Queue()

```

```
    level[v] = 0
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (level[k] == -1):
                level[k] = level[j]+1
                parent[k] = j
                q.addq(k)

    return(level,parent)
AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
print(BFSListPathLevel(AList,0))
```

Output

```
({0: 0, 1: 1, 2: 1, 3: 2, 4: 2}, {0: -1, 1: 0, 2: 0, 3: 1, 4: 1})
```

Depth First Search(DFS)

DFS for adjacency list of graph

DFS using Stack for adjacency list of graph

```
class Stack:
    def __init__(self):
        self.stack = []
    def Push(self,v):
        self.stack.append(v)
    def isempty(self):
        return(self.stack == [])
    def Pop(self):
        v = None
        if not self.isempty():
            v = self.stack.pop()
        return(v)
    def __str__(self):
        return(str(self.stack))

def DFSList(AList,v):
    visited = {}
    for i in AList.keys():
```

```
            visited[i] = False
        st = Stack()
        st.Push(v)
        while(not st.isempty()):
            j = st.Pop()
            if visited[j] == False:
                visited[j] = True
                for k in AList[j][::-1]:
                    if(not visited[k]):
                        st.Push(k)
        return(visited)
AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
print(DFSList(AList,0))
```

Output

```
{0: True, 1: True, 2: True, 3: True, 4: True}
```

DFS Recursive (without using external stack)

```
def DFSInitList(AList):
    # Initialization
    (visited,parent) = ({},{})
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    return(visited,parent)

def DFSList(AList,visited,parent,v):
    visited[v] = True
    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            (visited,parent) = DFSList(AList,visited,parent,k)

    return(visited,parent)
AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
v,p = DFSInitList(AList)
print(DFSList(AList,v,p,0))
```

Output

```
({0: True, 1: True, 2: True, 3: True, 4: True}, {0: -1, 1: 0, 2: 0, 3: 1, 4: 3})
```

DFS global for adjacency list of graph

```
(visited,parent) = ({},{})

def DFSInitListGlobal(AList):
```

```
    # Initialization
    for i in AList.keys():
        visited[i] = False
        parent[i] = -1
    return

def DFSListGlobal(AList,v):
    visited[v] = True

    for k in AList[v]:
        if (not visited[k]):
            parent[k] = v
            DFSListGlobal(AList,k)
    return
AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
DFSInitListGlobal(AList)
DFSListGlobal(AList,0)
print(visited,parent)
```

Output

```
{0: True, 1: True, 2: True, 3: True, 4: True} {0: -1, 1: 0, 2: 0, 3: 1, 4: 3}
```

DFS for adjacency matrix of graph

```
def neighbours(AMat,i):
    nbrs = []
    (rows,cols) = AMat.shape
    for j in range(cols):
        if AMat[i,j] == 1:
            nbrs.append(j)
    return(nbrs)
def DFSInit(AMat):
    # Initialization
    (rows,cols) = AMat.shape
    (visited,parent) = ({},{})
    for i in range(rows):
        visited[i] = False
        parent[i] = -1
    return(visited,parent)

def DFS(AMat,visited,parent,v):
    visited[v] = True

    for k in neighbours(AMat,v):
        if (not visited[k]):
            parent[k] = v
            (visited,parent) = DFS(AMat,visited,parent,k)

    return(visited,parent)
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
```

```python
size = len(V)
import numpy as np
AMat = np.zeros(shape=(size,size))
for (i,j) in E:
    AMat[i,j] = 1
v,p=DFSInit(AMat)
print(DFS(AMat,v,p,0))
```

Output

```
({0: True, 1: True, 2: True, 3: True, 4: True}, {0: -1, 1: 0, 2: 0, 3: 1, 4:
3})
```

DFS global for adjacency matrix of graph

```python
(visited,parent) = ({},{})
def neighbours(AMat,i):
    nbrs = []
    (rows,cols) = AMat.shape
    for j in range(cols):
        if AMat[i,j] == 1:
            nbrs.append(j)
    return(nbrs)

def DFSInitGlobal(AMat):
    # Initialization
    (rows,cols) = AMat.shape
    for i in range(rows):
        visited[i] = False
        parent[i] = -1
    return

def DFSGlobal(AMat,v):
    visited[v] = True

    for k in neighbours(AMat,v):
        if (not visited[k]):
            parent[k] = v
            DFSGlobal(AMat,k)

    return
V = [0,1,2,3,4]
E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
size = len(V)
import numpy as np
AMat = np.zeros(shape=(size,size))
for (i,j) in E:
    AMat[i,j] = 1
DFSInitGlobal(AMat)
DFSGlobal(AMat,0)
print(visited,parent)
```

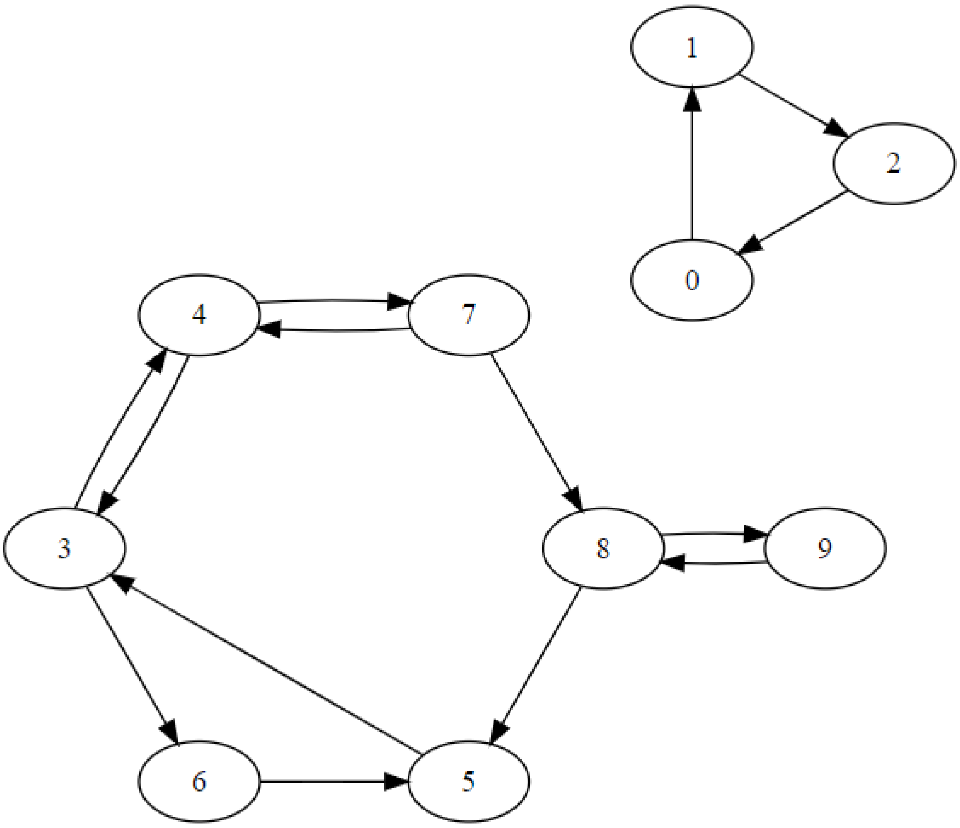
Output

```
{0: True, 1: True, 2: True, 3: True, 4: True} {0: -1, 1: 0, 2: 0, 3: 1, 4: 3}
```

Application of BFS and DFS

Find Components using BFS

For given graph




```
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))

```

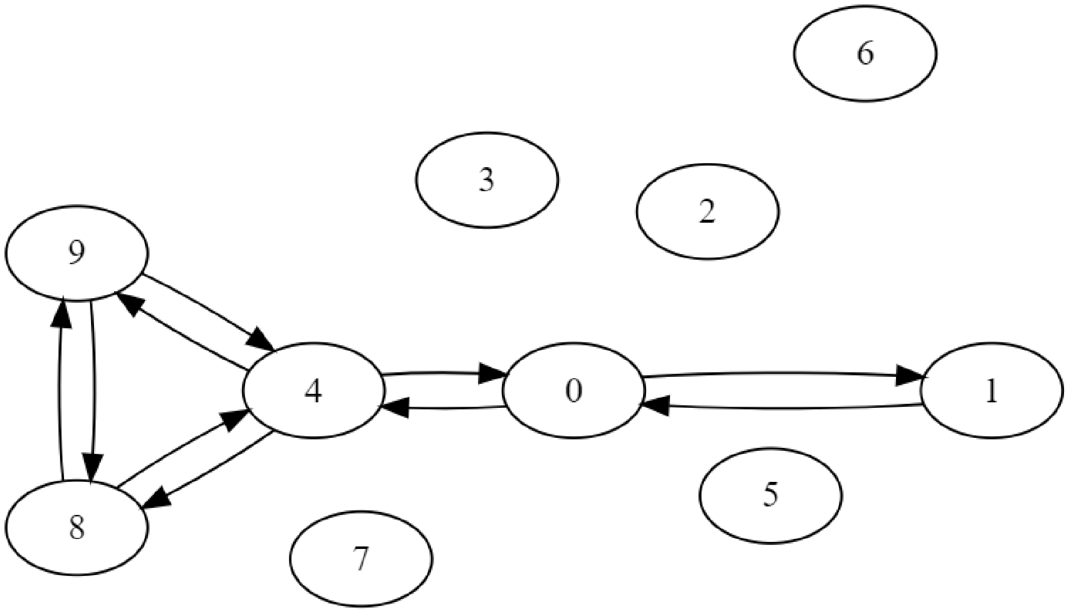
```python
def BFSList(AList,v):
    visited = {}
    for i in AList.keys():
        visited[i] = False
    q = Queue()

    visited[v] = True
    q.addq(v)

    while(not q.isempty()):
        j = q.delq()
        for k in AList[j]:
            if (not visited[k]):
                visited[k] = True
                q.addq(k)
    return(visited)
def Components(AList):
    component = {}
    for i in AList.keys():
        component[i] = -1
    (compid,seen) = (0,0)
    while seen < max(AList.keys()):
        startv = min([i for i in AList.keys() if component[i] == -1])
        visited = BFSList(AList,startv)
        for i in visited.keys():
            if visited[i]:
                seen = seen + 1
                component[i] = compid
        compid = compid + 1
    return(component)
AList = {0: [1], 1: [2], 2: [0], 3: [4, 6], 4: [3, 7], 5: [3, 7], 6: [5], 7: [4, 8], 8: [5, 9], 9: [8]}
print(Components(AList))
```

Output

```
{0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 1, 9: 1}
```

Pre Post using DFS

For given graph




```python
(visited,pre,post) = ({},{},{})

def DFSInitPrePost(AList):
    # Initialization
    for i in AList.keys():
        visited[i] = False
        (pre[i],post[i]) = (-1,-1)
    return

def DFSListPrePost(AList,v,count):
    visited[v] = True
    pre[v] = count
    count = count+1
    for k in AList[v]:
        if (not visited[k]):
            count = DFSListPrePost(AList,k,count)
    post[v] = count
    count = count+1
    return(count)
AList = {0: [1, 4],1: [0],2: [],3: [],4: [0, 8, 9],5: [],6: [],7: [],8: [4, 9],9: [8, 4]}
DFSInitPrePost(AList)
print(DFSListPrePost(AList,0,0))
print(visited)
print(pre)
print(post)
```

Output

```
10
{0: True, 1: True, 2: False, 3: False, 4: True, 5: False, 6: False, 7: False,
8: True, 9: True}
{0: 0, 1: 1, 2: -1, 3: -1, 4: 3, 5: -1, 6: -1, 7: -1, 8: 4, 9: 5}
{0: 9, 1: 2, 2: -1, 3: -1, 4: 8, 5: -1, 6: -1, 7: -1, 8: 7, 9: 6}
```

Directed Acyclic Graph(DAG)

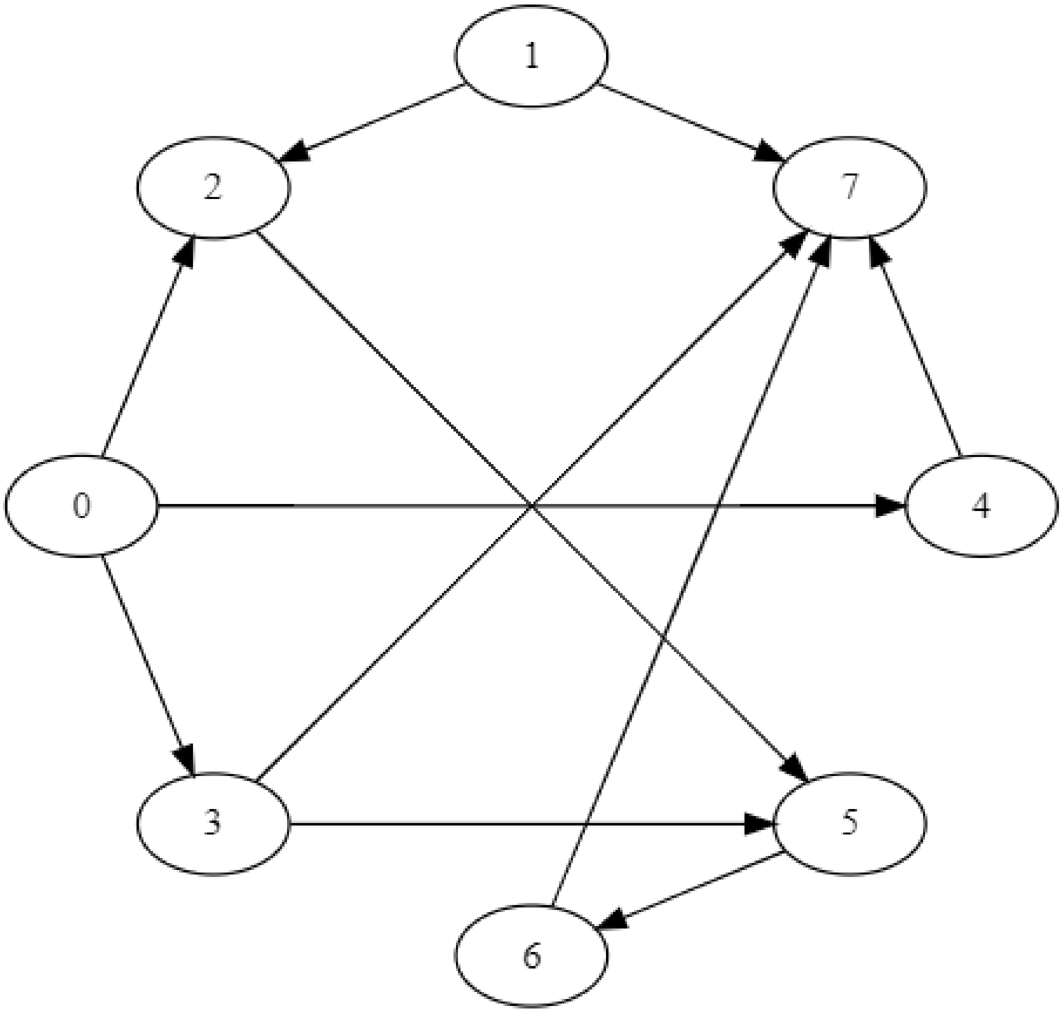
A Directed Acyclic Graph(DAG) is a directed graph with no directed cycles. That is, it consists of vertices and edges, with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

Directed acyclic graphs are a natural way to represent dependencies

- Arise in many contexts
  - Pre-requisites between courses for completing a degree
  - Recipe for cooking
  - Construction project
- Problems to be solved on DAGS
  - Topological sorting
  - Longest paths

Topological Sort

For given graph(DAG)


1
2
7
0
4
3
5
6

Topological sort for Adjacency matrix

```python
def toposort(AMat):
    (rows,cols) = AMat.shape
    indegree = {}
    toposortlist = []
    for c in range(cols):
        indegree[c] = 0
        for r in range(rows):
            if AMat[r,c] == 1:
                indegree[c] = indegree[c] + 1

    for i in range(rows):
        j = min([k for k in range(cols)
                  if indegree[k] == 0])
        toposortlist.append(j)
        indegree[j] = indegree[j]-1
        for k in range(cols):
            if AMat[j,k] == 1:
                indegree[k] = indegree[k] - 1
    return(toposortlist)
edges=[(0,2),(0,3),(0,4),(1,2),(1,7),(2,5),(3,5),(3,7),(4,7),(5,6),(6,7)]
size = 8
import numpy as np
```

```python
AMat = np.zeros(shape=(size,size))
for (i,j) in edges:
    AMat[i,j] = 1
print(toposort(AMat))
```

Output

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Topological sort for Adjacency list

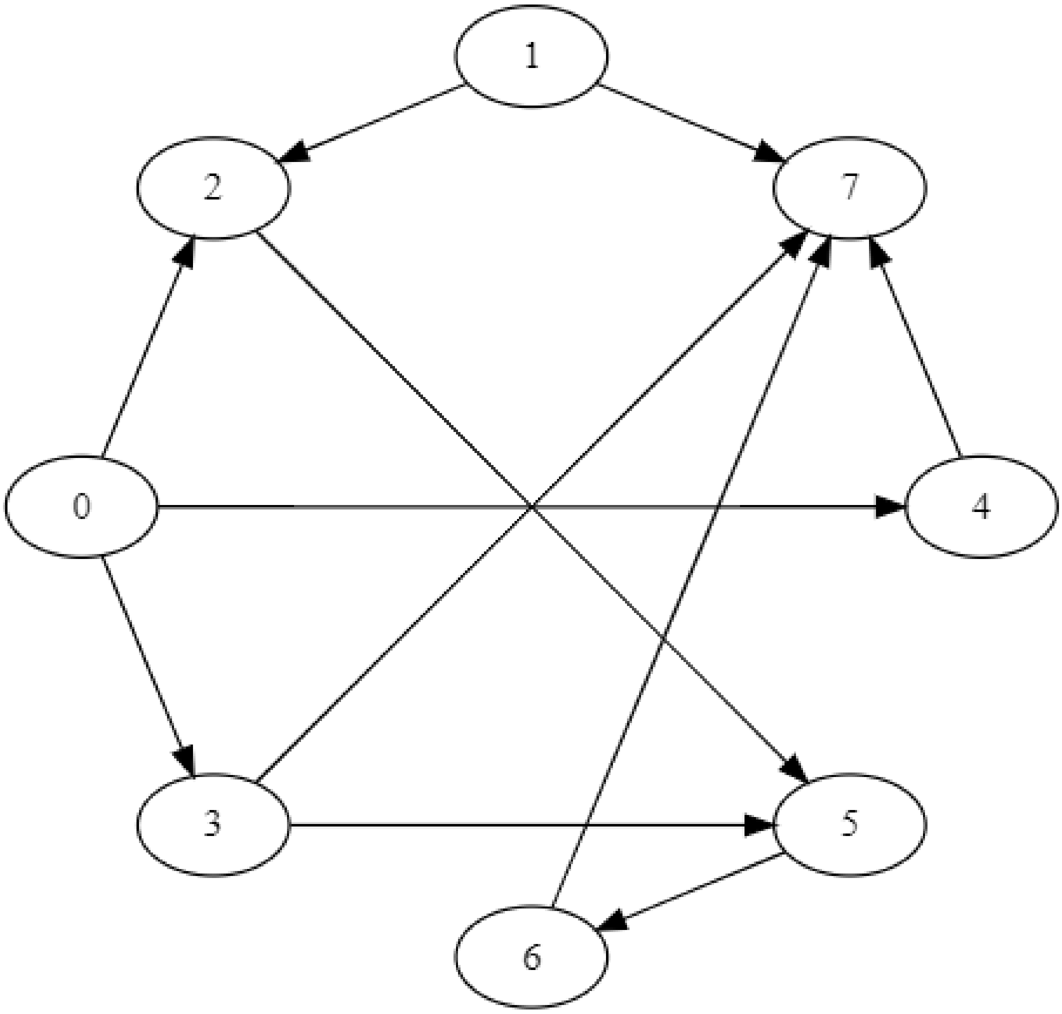
```python
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))
def toposortlist(AList):
    (indegree,toposortlist) = ({},[])
    zerodegreeq = Queue()

    for u in AList.keys():
        indegree[u] = 0

    for u in AList.keys():
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        toposortlist.append(j)
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            if indegree[k] == 0:
                zerodegreeq.addq(k)
    return(toposortlist)
AList={0: [2, 3, 4], 1: [2, 7], 2: [5], 3: [5, 7], 4: [7], 5: [6], 6: [7],
7: []}
print(toposortlist(AList))
```

Output

```
[0, 1, 3, 4, 2, 5, 6, 7]
```

Longest Path in DAG

For given graph(DAG)



```python
class Queue:
    def __init__(self):
        self.queue = []
    def addq(self,v):
        self.queue.append(v)
    def isempty(self):
        return(self.queue == [])
    def delq(self):
        v = None
        if not self.isempty():
            v = self.queue[0]
            self.queue = self.queue[1:]
        return(v)
    def __str__(self):
        return(str(self.queue))
def longestpathlist(AList):
    (indegree,lpath) = ({},{})
    zerodegreeq = Queue()

    for u in AList.keys():
        (indegree[u],lpath[u]) = (0,0)

    for u in AList.keys():
```

```
        for v in AList[u]:
            indegree[v] = indegree[v] + 1

    for u in AList.keys():
        if indegree[u] == 0:
            zerodegreeq.addq(u)

    while (not zerodegreeq.isempty()):
        j = zerodegreeq.delq()
        indegree[j] = indegree[j]-1
        for k in AList[j]:
            indegree[k] = indegree[k] - 1
            lpath[k] = max(lpath[k],lpath[j]+1)
            if indegree[k] == 0:
                zerodegreeq.addq(k)

    return(lpath)
AList={0: [2, 3, 4], 1: [2, 7], 2: [5], 3: [5, 7], 4: [7], 5: [6], 6: [7], 7: []}
print(longestpathlist(AList))
```

Output

```
{0: 0, 1: 0, 2: 1, 3: 1, 4: 1, 5: 2, 6: 3, 7: 4}
```

Visualization of graph algorithms

VISUALGO.NET / en /dfsbfs

GRAPH TRAVERSAL (DFS/BFS)

LOGIN

<

Source - https://visualgo.net/en/dfsbfs