



IIT Madras
BSc Degree

[Home](#)[Week-3](#)[Week-5](#)

PDSA - Week 4

PDSA - Week 4

- Introduction of Graph
- Types of graph
- Graph Representation(Unweighted)
 - For directed graph
 - For undirected graph
- Graph Traversing Algorithm
 - Breadth First Search(BFS)
 - BFS for adjacency list of graph
 - BFS for adjacency matrix of graph
 - Depth First Search(DFS)
 - DFS for adjacency list of graph
 - DFS for adjacency matrix of graph
 - Application of BFS and DFS
- Directed Acyclic Graph(DAG)
 - Topological Sort
 - Longest Path in DAG
- Visualization of graph algorithms

Introduction of Graph

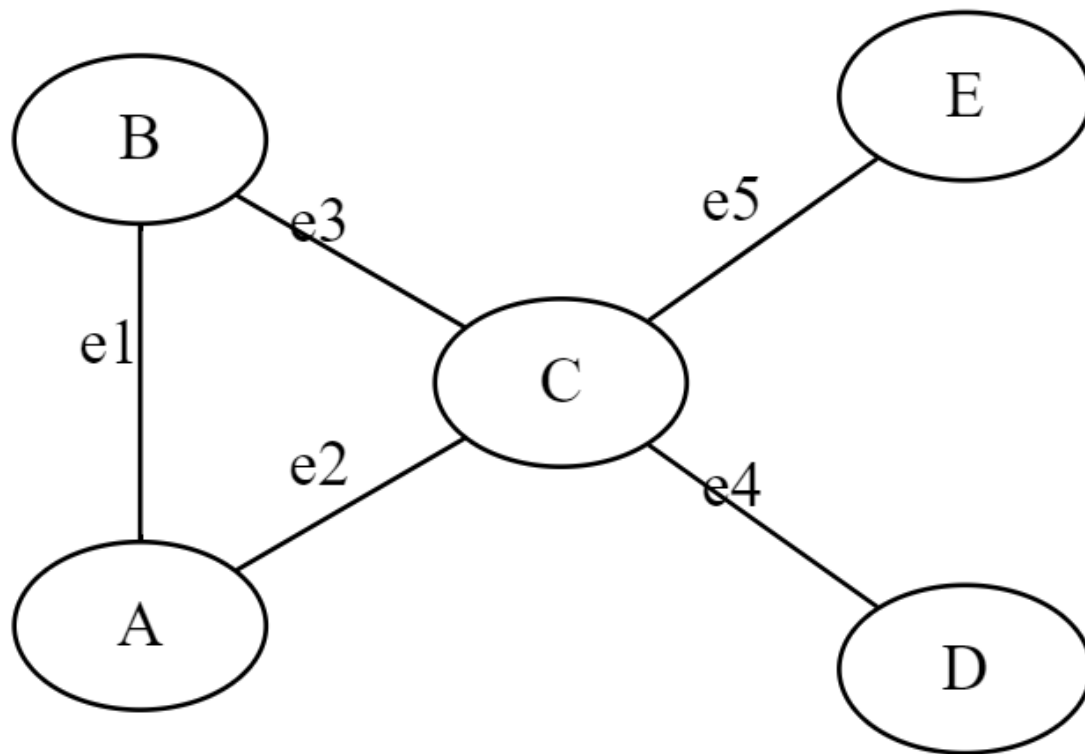
Graph:- It is a non-linear data structure. A graph G consist of a non empty set V where members are called the vertices of graph and the set E where member are called the edges.

$G = (V, E)$

V = set of vertices

E = set of edges

Example:-

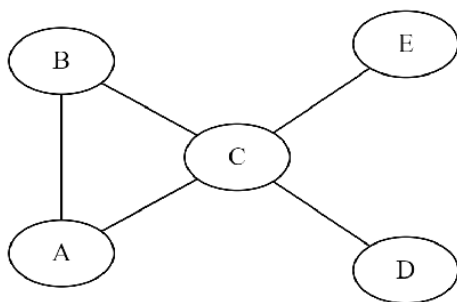


$G = (V, E)$

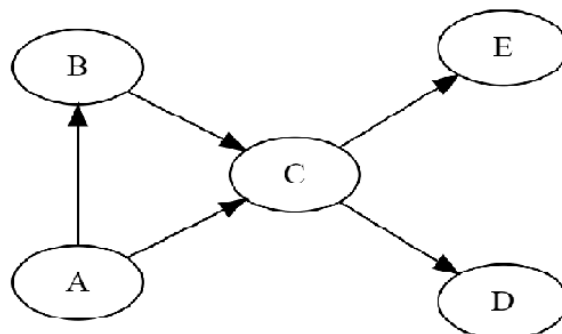
$V = \{A, B, C, D, E\}$

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

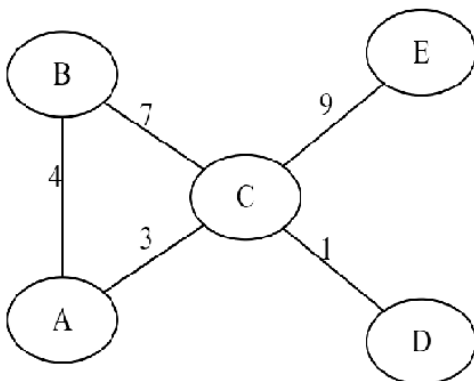
Types of graph



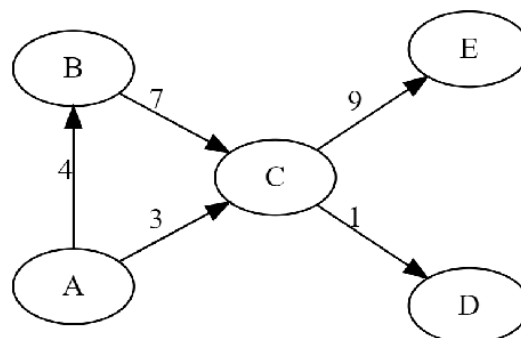
Undirected and unweighted graph



Directed and unweighted graph



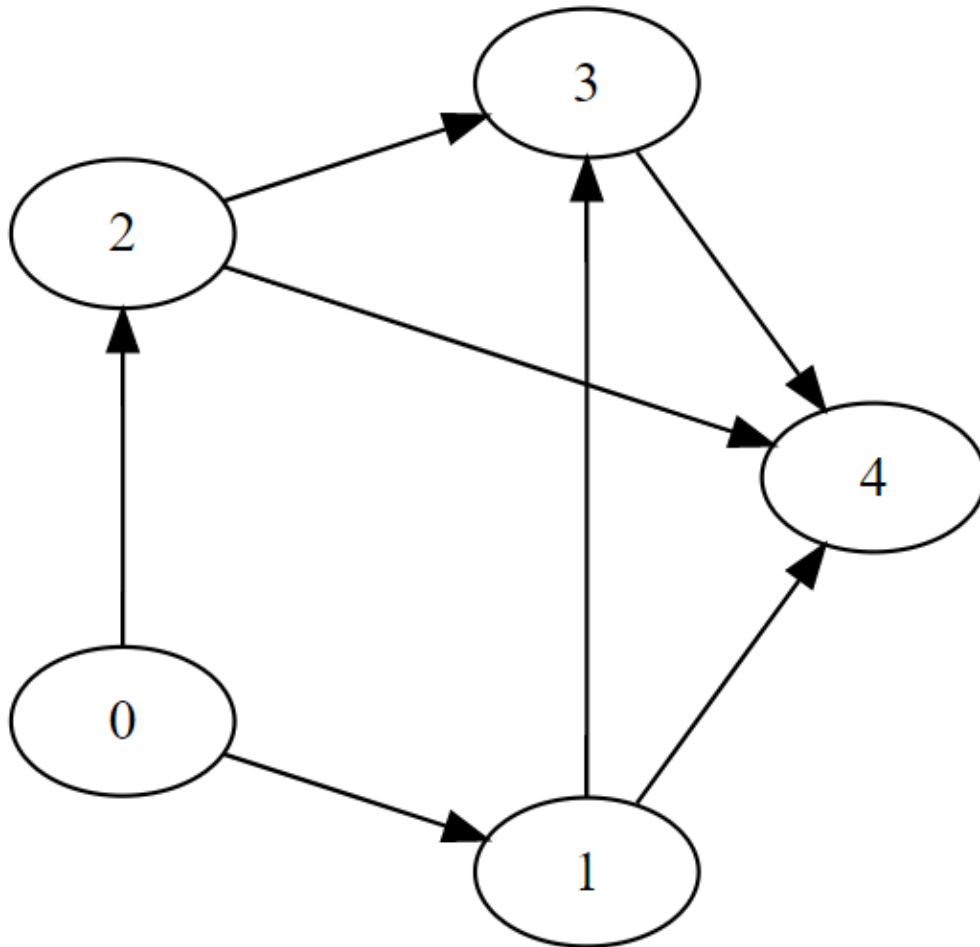
Undirected and weighted graph



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

```

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

```

Output adjacency matrix (AMat)

```

1  [[0. 1. 1. 0. 0.]
2   [0. 0. 0. 1. 1.]
3   [0. 0. 0. 1. 1.]
4   [0. 0. 0. 0. 1.]
5   [0. 0. 0. 0. 0.]]
6  # AMat[i,j] == 1 represent edge from i to j

```

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

```

1  V = [0,1,2,3,4]
2  E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
3  size = len(V)
4  AMat = []
5  for i in range(size):
6      row = []
7      for j in range(size):
8          row.append(0)
9      AMat.append(row.copy())
10 for (i,j) in E:
11     AMat[i][j] = 1 # mark 1 if edge present in graph from i to j , otherwise
12     0
13 print(AMat)

```

Output adjacency matrix (AMat)

```

1  [[0, 1, 1, 0, 0],
2   [0, 0, 0, 1, 1],
3   [0, 0, 0, 1, 1],
4   [0, 0, 0, 0, 1],
5   [0, 0, 0, 0, 0]]
6  # AMat[i][j] == 1 represent edge from i to j

```

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

```

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

```

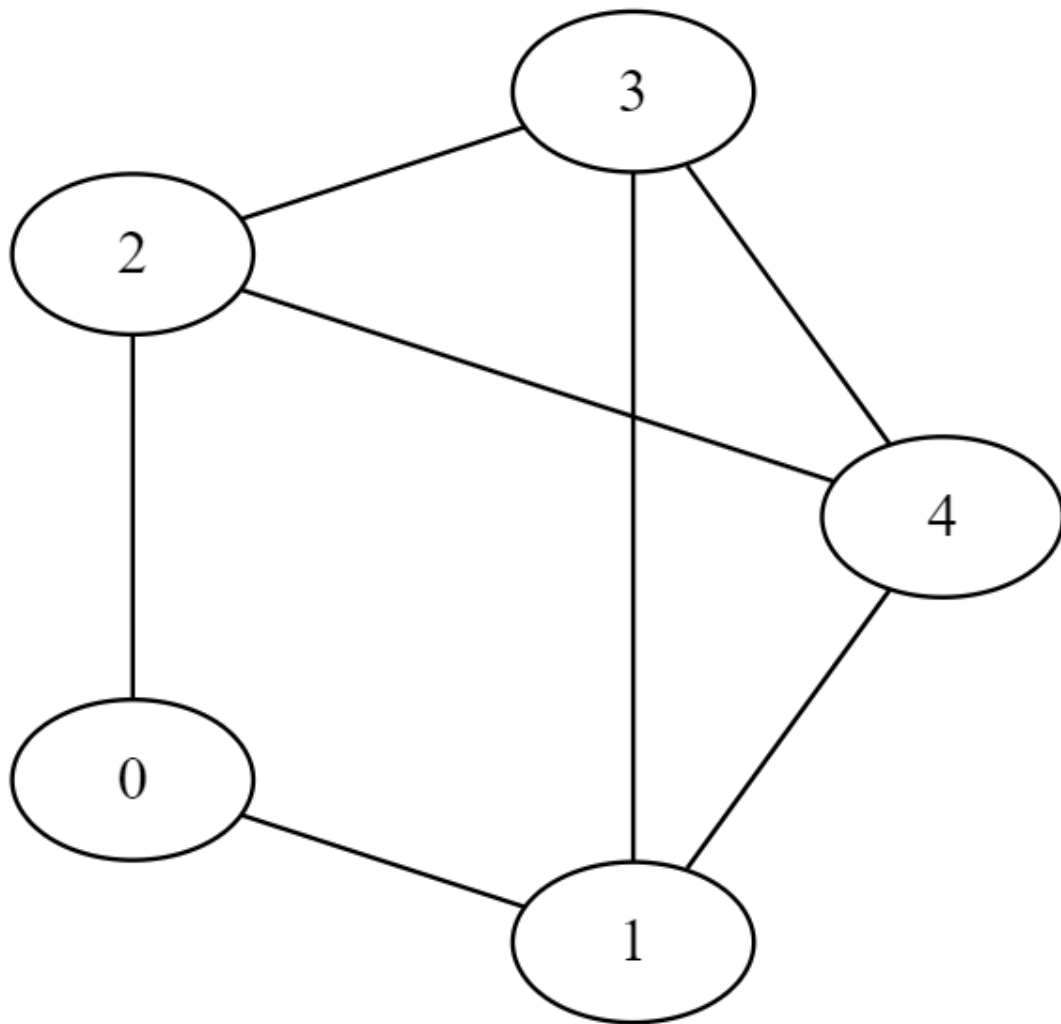
Output adjacency list (AList)

```

1  {0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
2  # 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

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

Output adjacency matrix (AMat)

```

1  [[0. 1. 1. 0. 0.]
2   [1. 0. 0. 1. 1.]
3   [1. 0. 0. 1. 1.]
4   [0. 1. 1. 0. 1.]
5   [0. 1. 1. 1. 0.]]
6  # AMat[i,j] == 1 represent edge from i to j

```

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

```

1  v = [0,1,2,3,4]
2  E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
3  UE = E + [ (j,i) for (i,j) in E] # each edge represented by two tuple (u,v)
   and (v,u)
4  size = len(v)
5  AMat = []
6  for i in range(size):
7      row = []
8      for j in range(size):
9          row.append(0)
10     AMat.append(row.copy())
11     for (i,j) in UE:
12         AMat[i][j] = 1 # mark 1 if edge present in graph from i to j , otherwise
   0
13     print(AMat)

```

Output adjacency matrix (AMat)

```

1  [[0, 1, 1, 0, 0],
2   [1, 0, 0, 1, 1],
3   [1, 0, 0, 1, 1],
4   [0, 1, 1, 0, 1],
5   [0, 1, 1, 1, 0]]
6  # AMat[i][j] == 1 represent edge from i to j

```

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

```

1  v = [0,1,2,3,4]
2  E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
3  UE = E + [ (j,i) for (i,j) in E] # each edge represented by two tuple (u,v)
   and (v,u)
4  size = len(v)
5  AList = {}
6  # In dictionary AList, for example, AList[i] = [j,k] represent two edge from
   i to j and i to k
7  for i in range(size):
8      AList[i] = []
9      for (i,j) in UE:
10         AList[i].append(j)
11     print(AList)

```

Output adjacency list (AList)

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

Graph Traversing Algorithm

Breadth First Search(BFS)

Breadth First Search

Working Concept

BFS for adjacency list of graph

```
1 class Queue:
2     def __init__(self):
3         self.queue = []
4     def addq(self,v):
5         self.queue.append(v)
6     def isempty(self):
7         return(self.queue == [])
8     def delq(self):
```

```

9         v = None
10        if not self.isempty():
11            v = self.queue[0]
12            self.queue = self.queue[1:]
13        return(v)
14    def __str__(self):
15        return(str(self.queue))
16
17    def BFSList(AList,v):
18        visited = {}
19        for i in AList.keys():
20            visited[i] = False
21        q = Queue()
22
23        visited[v] = True
24        q.addq(v)
25
26        while(not q.isempty()):
27            j = q.delq()
28            for k in AList[j]:
29                if (not visited[k]):
30                    visited[k] = True
31                    q.addq(k)
32        return(visited)
33    AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
34    print(BFSList(AList,0))

```

Output

```

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

```

BFS for adjacency matrix of graph

```

1    class Queue:
2        def __init__(self):
3            self.queue = []
4        def addq(self,v):
5            self.queue.append(v)
6        def isempty(self):
7            return(self.queue == [])
8        def delq(self):
9            v = None
10           if not self.isempty():
11               v = self.queue[0]
12               self.queue = self.queue[1:]
13           return(v)
14        def __str__(self):
15            return(str(self.queue))
16
17    def neighbours(AMat,i):
18        nbrs = []
19        (rows,cols) = AMat.shape

```



```

20     for j in range(cols):
21         if AMat[i,j] == 1:
22             nbrs.append(j)
23     return(nbrs)
24 def BFS(AMat,v):
25     (rows,cols) = AMat.shape
26     visited = {}
27     for i in range(rows):
28         visited[i] = False
29     q = Queue()
30
31     visited[v] = True
32     q.addq(v)
33
34     while(not q.isempty()):
35         j = q.delq()
36         for k in neighbours(AMat,j):
37             if (not visited[k]):
38                 visited[k] = True
39                 q.addq(k)
40
41     return(visited)
42
43 v = [0,1,2,3,4]
44 E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
45 size = len(v)
46 import numpy as np
47 AMat = np.zeros(shape=(size,size))
48 for (i,j) in E:
49     AMat[i,j] = 1
50 print(BFS(AMat,0))

```

Output

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

List of path from vertex **v** using BFS

```

1 class Queue:
2     def __init__(self):
3         self.queue = []
4     def addq(self,v):
5         self.queue.append(v)
6     def isempty(self):
7         return(self.queue == [])
8     def delq(self):
9         v = None
10        if not self.isempty():
11            v = self.queue[0]
12            self.queue = self.queue[1:]
13        return(v)
14    def __str__(self):
15        return(str(self.queue))

```

```

16
17 def BFSListPath(AList,v):
18     (visited,parent) = ({},{})
19     for i in AList.keys():
20         visited[i] = False
21         parent[i] = -1
22     q = Queue()
23
24     visited[v] = True
25     q.addq(v)
26
27     while(not q.isempty()):
28         j = q.delq()
29         for k in AList[j]:
30             if (not visited[k]):
31                 visited[k] = True
32                 parent[k] = j
33                 q.addq(k)
34
35     return(visited,parent)
36 AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
37 print(BFSListPath(AList,0))

```

Output

```

1 ({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

```

1 class Queue:
2     def __init__(self):
3         self.queue = []
4     def addq(self,v):
5         self.queue.append(v)
6     def isempty(self):
7         return(self.queue == [])
8     def delq(self):
9         v = None
10        if not self.isempty():
11            v = self.queue[0]
12            self.queue = self.queue[1:]
13        return(v)
14    def __str__(self):
15        return(str(self.queue))
16
17 def BFSListPathLevel(AList,v):
18     (level,parent) = ({},{})
19     for i in AList.keys():
20         level[i] = -1
21         parent[i] = -1
22     q = Queue()
23

```

```
24     level[v] = 0
25     q.addq(v)
26
27     while(not q.isempty()):
28         j = q.delq()
29         for k in AList[j]:
30             if (level[k] == -1):
31                 level[k] = level[j]+1
32                 parent[k] = j
33                 q.addq(k)
34
35     return(level,parent)
36 AList = {0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
37 print(BFSListPathLevel(AList,0))
```

Output

```
1 | ({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

```
1 class Stack:
2     def __init__(self):
3         self.stack = []
4     def Push(self,v):
5         self.stack.append(v)
6     def isempty(self):
7         return(self.stack == [])
8     def Pop(self):
9         v = None
10        if not self.isempty():
11            v = self.stack.pop()
12        return(v)
13    def __str__(self):
14        return(str(self.stack))
15
16 def DFSList(AList,v):
17     visited = {}
18     for i in AList.keys():
```

```

19     visited[i] = False
20     st = Stack()
21     st.Push(v)
22     while(not st.isempty()):
23         j = st.Pop()
24         if visited[j] == False:
25             visited[j] = True
26             for k in AList[j][::-1]:
27                 if(not visited[k]):
28                     st.Push(k)
29     return(visited)
30 AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
31 print(DFSList(AList,0))

```

Output

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

DFS Recursive (without using external stack)

```

1  def DFSInitList(AList):
2      # Initialization
3      (visited,parent) = ({},{})
4      for i in AList.keys():
5          visited[i] = False
6          parent[i] = -1
7      return(visited,parent)
8
9  def DFSList(AList,visited,parent,v):
10     visited[v] = True
11     for k in AList[v]:
12         if (not visited[k]):
13             parent[k] = v
14             (visited,parent) = DFSList(AList,visited,parent,k)
15
16     return(visited,parent)
17 AList ={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
18 v,p = DFSInitList(AList)
19 print(DFSList(AList,v,p,0))

```

Output

```
1 | ({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

```

1  (visited,parent) = ({},{})
2
3  def DFSInitListGlobal(AList):

```

```

4      # Initialization
5      for i in AList.keys():
6          visited[i] = False
7          parent[i] = -1
8      return
9
10     def DFSListGlobal(AList,v):
11         visited[v] = True
12
13         for k in AList[v]:
14             if (not visited[k]):
15                 parent[k] = v
16                 DFSListGlobal(AList,k)
17         return
18     AList={0: [1, 2], 1: [3, 4], 2: [4, 3], 3: [4], 4: []}
19     DFSInitListGlobal(AList)
20     DFSListGlobal(AList,0)
21     print(visited,parent)

```

Output

```
1 | {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

```

1     def neighbours(AMat,i):
2         nbrs = []
3         (rows,cols) = AMat.shape
4         for j in range(cols):
5             if AMat[i,j] == 1:
6                 nbrs.append(j)
7         return(nbrs)
8     def DFSInit(AMat):
9         # Initialization
10        (rows,cols) = AMat.shape
11        (visited,parent) = ({},{})
12        for i in range(rows):
13            visited[i] = False
14            parent[i] = -1
15        return(visited,parent)
16
17     def DFS(AMat,visited,parent,v):
18         visited[v] = True
19
20         for k in neighbours(AMat,v):
21             if (not visited[k]):
22                 parent[k] = v
23                 (visited,parent) = DFS(AMat,visited,parent,k)
24
25         return(visited,parent)
26     v = [0,1,2,3,4]
27     E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]

```

```

28 size = len(V)
29 import numpy as np
30 AMat = np.zeros(shape=(size,size))
31 for (i,j) in E:
32     AMat[i,j] = 1
33 v,p=DFSInit(AMat)
34 print(DFS(AMat,v,p,0))

```

Output

```

1 | ({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

```

1  (visited,parent) = ({},{})
2  def neighbours(AMat,i):
3      nbrs = []
4      (rows,cols) = AMat.shape
5      for j in range(cols):
6          if AMat[i,j] == 1:
7              nbrs.append(j)
8      return(nbrs)
9
10 def DFSInitGlobal(AMat):
11     # Initialization
12     (rows,cols) = AMat.shape
13     for i in range(rows):
14         visited[i] = False
15         parent[i] = -1
16     return
17
18 def DFSGlobal(AMat,v):
19     visited[v] = True
20
21     for k in neighbours(AMat,v):
22         if (not visited[k]):
23             parent[k] = v
24             DFSGlobal(AMat,k)
25
26     return
27 v = [0,1,2,3,4]
28 E = [(0, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 3), (3, 4)]
29 size = len(V)
30 import numpy as np
31 AMat = np.zeros(shape=(size,size))
32 for (i,j) in E:
33     AMat[i,j] = 1
34 DFSInitGlobal(AMat)
35 DFSGlobal(AMat,0)
36 print(visited,parent)

```

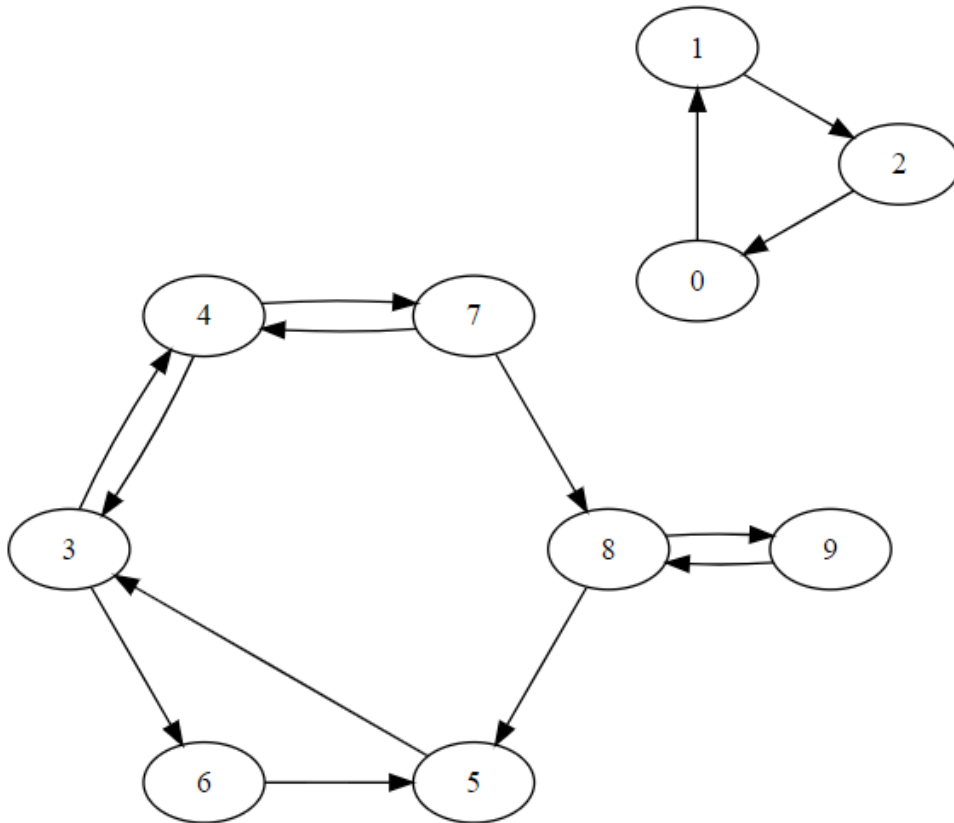
Output

```
1 | {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



```

1  class Queue:
2      def __init__(self):
3          self.queue = []
4      def addq(self,v):
5          self.queue.append(v)
6      def isempty(self):
7          return(self.queue == [])
8      def delq(self):
9          v = None
10         if not self.isempty():
11             v = self.queue[0]
12             self.queue = self.queue[1:]
13         return(v)
14     def __str__(self):
15         return(str(self.queue))
16 
```



```

17 def BFSList(AList,v):
18     visited = {}
19     for i in AList.keys():
20         visited[i] = False
21     q = Queue()
22
23     visited[v] = True
24     q.addq(v)
25
26     while(not q.isempty()):
27         j = q.delq()
28         for k in AList[j]:
29             if (not visited[k]):
30                 visited[k] = True
31                 q.addq(k)
32     return(visited)
33 def Components(AList):
34     component = {}
35     for i in AList.keys():
36         component[i] = -1
37     (compid,seen) = (0,0)
38     while seen < max(AList.keys()):
39         startv = min([i for i in AList.keys() if component[i] == -1])
40         visited = BFSList(AList,startv)
41         for i in visited.keys():
42             if visited[i]:
43                 seen = seen + 1
44                 component[i] = compid
45             compid = compid + 1
46     return(component)
47 AList = {0: [1], 1: [2], 2: [0], 3: [4, 6], 4: [3, 7], 5: [3, 7], 6: [5], 7:
48         [4, 8], 8: [5, 9], 9: [8]}
49 print(Components(AList))

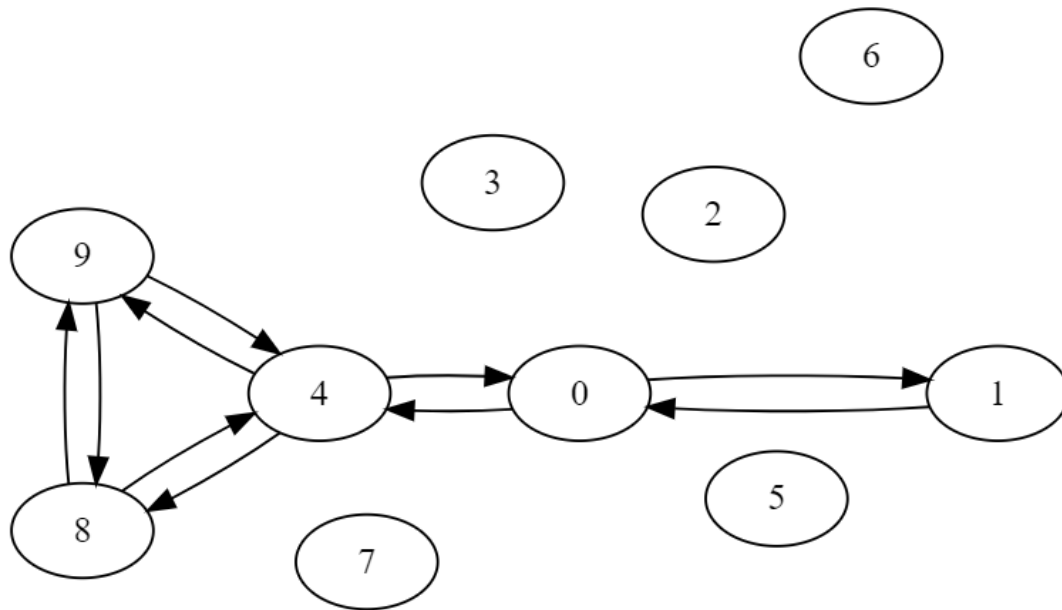
```

Output

```
1 | {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



```

1  (visited,pre,post) = ({},{},{})
2
3  def DFSInitPrePost(AList):
4      # Initialization
5      for i in AList.keys():
6          visited[i] = False
7          (pre[i],post[i]) = (-1,-1)
8      return
9
10 def DFSListPrePost(AList,v,count):
11     visited[v] = True
12     pre[v] = count
13     count = count+1
14     for k in AList[v]:
15         if (not visited[k]):
16             count = DFSListPrePost(AList,k,count)
17     post[v] = count
18     count = count+1
19     return(count)
20 AList = {0: [1, 4],1: [0],2: [],3: [],4: [0, 8, 9],5: [],6: [],7: [],8: [4,
21 9],9: [8, 4]}
22 DFSInitPrePost(AList)
23 print(DFSListPrePost(AList,0,0))
24 print(visited)
25 print(pre)
26 print(post)

```

Output

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

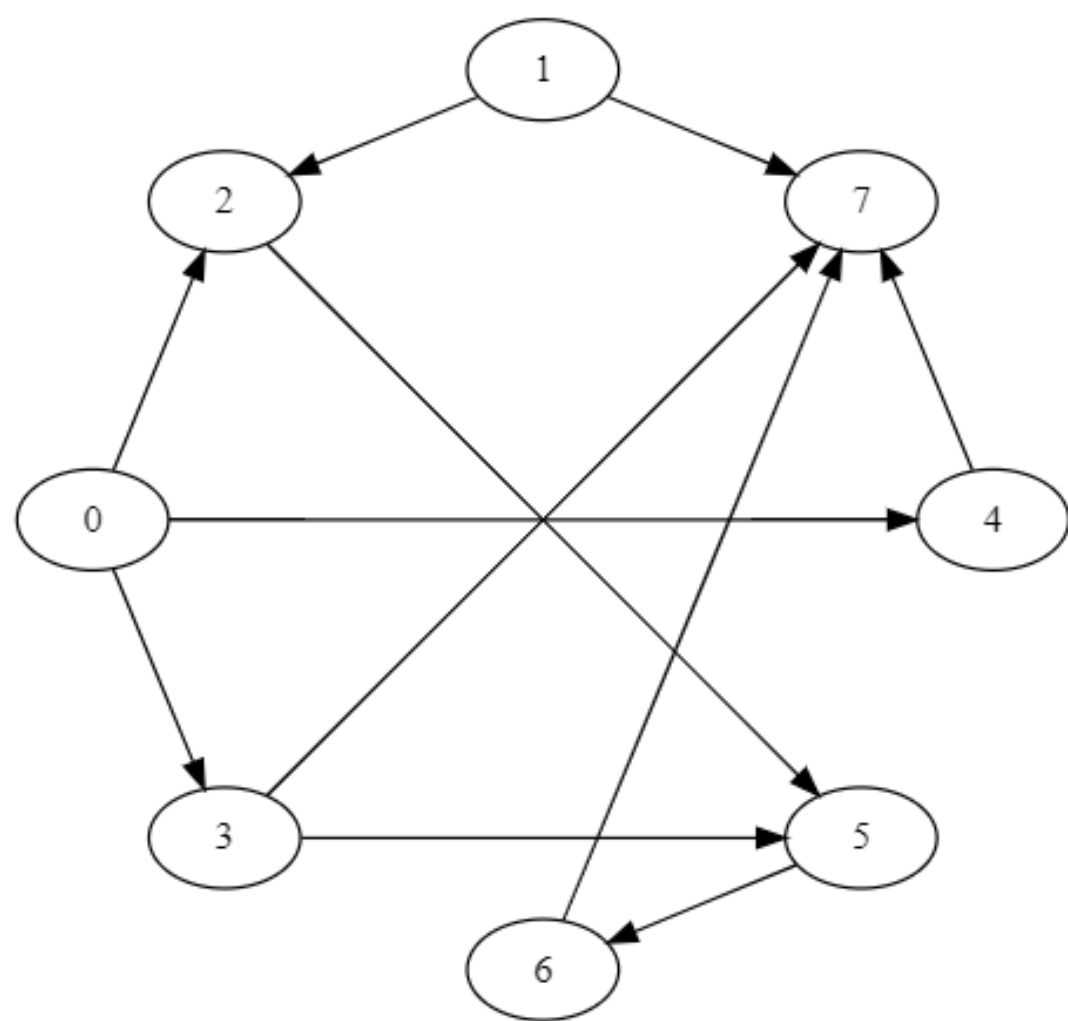
A Directed Acyclic Graph(DAG) is a directed graph with no directed cycles. That is, it consists of vertices and edges, with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

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)



Topological sort for Adjacency matrix

```
1 def toposort(AMat):
2     (rows,cols) = AMat.shape
3     indegree = {}
4     toposortlist = []
5     for c in range(cols):
6         indegree[c] = 0
7         for r in range(rows):
8             if AMat[r,c] == 1:
9                 indegree[c] = indegree[c] + 1
10
11     for i in range(rows):
12         j = min([k for k in range(cols)
13                 if indegree[k] == 0])
14         toposortlist.append(j)
15         indegree[j] = indegree[j]-1
16         for k in range(cols):
17             if AMat[j,k] == 1:
18                 indegree[k] = indegree[k] - 1
19     return(toposortlist)
20 edges=[(0,2),(0,3),(0,4),(1,2),(1,7),(2,5),(3,5),(3,7),(4,7),(5,6),(6,7)]
21 size = 8
22 import numpy as np
```

```

23 AMat = np.zeros(shape=(size,size))
24 for (i,j) in edges:
25     AMat[i,j] = 1
26 print(toposort(AMat))

```

Output

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

Topological sort for Adjacency list

```

1  class Queue:
2      def __init__(self):
3          self.queue = []
4      def addq(self,v):
5          self.queue.append(v)
6      def isempty(self):
7          return(self.queue == [])
8      def delq(self):
9          v = None
10         if not self.isempty():
11             v = self.queue[0]
12             self.queue = self.queue[1:]
13         return(v)
14     def __str__(self):
15         return(str(self.queue))
16 def toposortlist(AList):
17     (indegree,toposortlist) = ({},[])
18     zerodegreeeq = Queue()
19
20     for u in AList.keys():
21         indegree[u] = 0
22
23     for u in AList.keys():
24         for v in AList[u]:
25             indegree[v] = indegree[v] + 1
26
27     for u in AList.keys():
28         if indegree[u] == 0:
29             zerodegreeeq.addq(u)
30
31     while (not zerodegreeeq.isempty()):
32         j = zerodegreeeq.delq()
33         toposortlist.append(j)
34         indegree[j] = indegree[j]-1
35         for k in AList[j]:
36             indegree[k] = indegree[k] - 1
37             if indegree[k] == 0:
38                 zerodegreeeq.addq(k)
39     return(toposortlist)
40 AList={0: [2, 3, 4], 1: [2, 7], 2: [5], 3: [5, 7], 4: [7], 5: [6], 6: [7],
41       7: []}
42 print(toposortlist(AList))

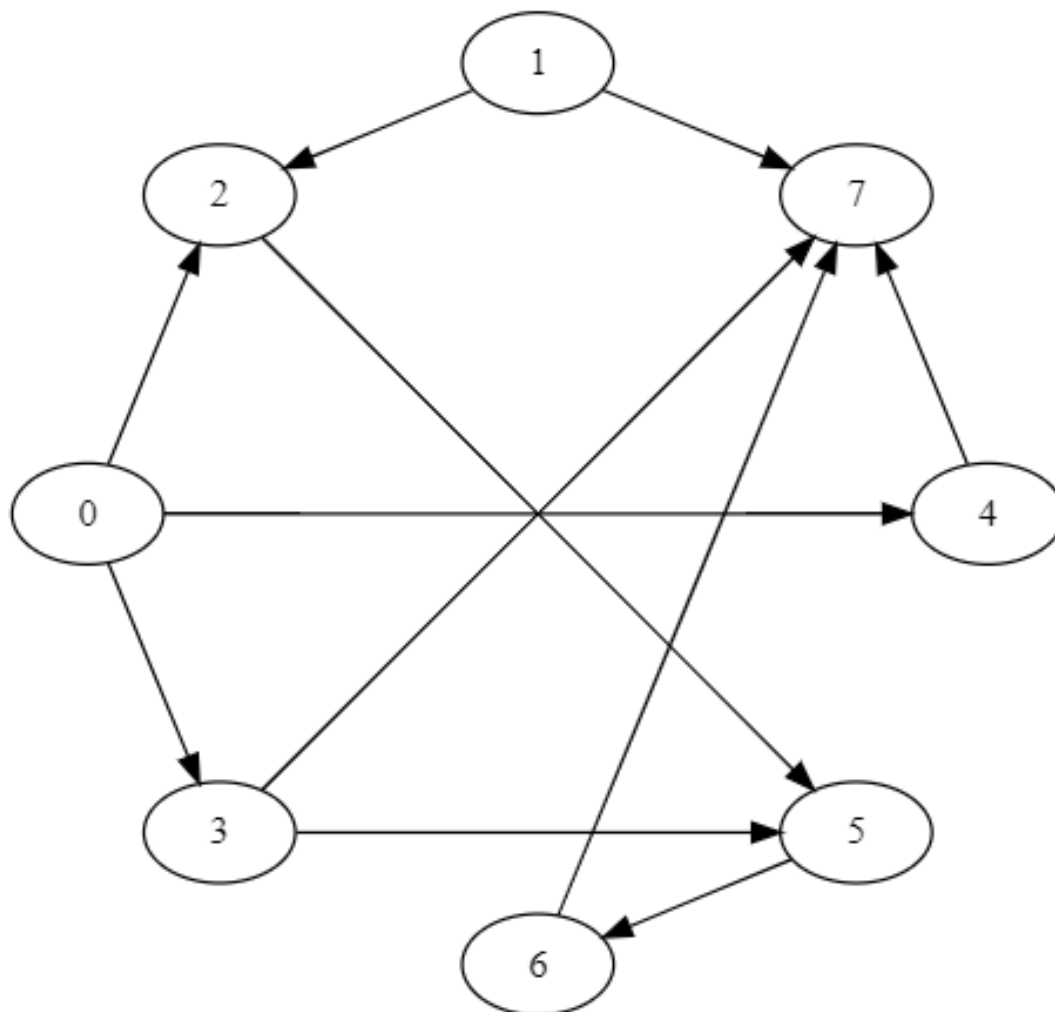
```

Output

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

Longest Path in DAG

For given graph(DAG)



```
1 class Queue:
2     def __init__(self):
3         self.queue = []
4     def addq(self,v):
5         self.queue.append(v)
6     def isempty(self):
7         return(self.queue == [])
8     def delq(self):
9         v = None
10        if not self.isempty():
11            v = self.queue[0]
12            self.queue = self.queue[1:]
13        return(v)
14    def __str__(self):
15        return(str(self.queue))
16 def longestpathlist(AList):
17     (indegree,lpath) = ({},{})
18     zerodegreeq = Queue()
19
20     for u in AList.keys():
21         (indegree[u],lpath[u]) = (0,0)
22
23     for u in AList.keys():
```



```
24     for v in AList[u]:
25         indegree[v] = indegree[v] + 1
26
27     for u in AList.keys():
28         if indegree[u] == 0:
29             zerodegreeeq.addq(u)
30
31     while (not zerodegreeeq.isempty()):
32         j = zerodegreeeq.delq()
33         indegree[j] = indegree[j]-1
34         for k in AList[j]:
35             indegree[k] = indegree[k] - 1
36             lpath[k] = max(lpath[k],lpath[j]+1)
37             if indegree[k] == 0:
38                 zerodegreeeq.addq(k)
39
40     return(lpath)
41 AList={0: [2, 3, 4], 1: [2, 7], 2: [5], 3: [5, 7], 4: [7], 5: [6], 6: [7],
42       7: []}
43 print(longestpathlist(AList))
```

Output

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

Visualization of graph algorithms

We use cookies to improve our website.

By clicking ACCEPT, you agree to our use of Google Analytics for analysing user behaviour and improving user experience as described in our Privacy Policy.

< By clicking reject, only cookies necessary

Source - <https://visualgo.net/en/dfsdfs>