Graph Data Structure in Python:

A graph is non-linear data structure that consists of the following two components

1. A finite set of vertices also called as nodes.
2. A finite set of edges in the form of (u,v) , u and v are two vertices in Graph

There are different terms are there related to Graphs

1. Directed Graph

A graph with directions are called as directed graph (one way).

2. Undirected Graph

A graph without any direction for edges are called as undirected graph (two way)

We can represent the graph in programming by using two ways.

1. Adjacency Matrix Representation
2. Adjacency List Representation

Ex:

```
def addEdge(L,u,v):
      L[u][v] = 1
      L[v][u] = 1

def printGraph(L):
      for i in range(len(L)):
            for j in range(len(L)):
                  print(L[i][j],end=' ')
            print()

v = 5
L = [[0]*v for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,4)
addEdge(L,1,4)
addEdge(L,1,3)
addEdge(L,1,2)
addEdge(L,2,3)
addEdge(L,3,4)
printGraph(L)
```

```
C:\test>py test.py
0 1 0 0 1
1 0 1 1 1
0 1 0 1 0
0 1 1 0 1
1 1 0 1 0
```

Ex:

```
def addEdge(L,u,v):
      L[u].append(v)
      L[v].append(u)

def printGraph(L):
      for i in range(len(L)):
```

```
		print(i,L[i])

v = 5
L = [[] for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,4)
addEdge(L,1,2)
addEdge(L,1,3)
addEdge(L,1,4)
addEdge(L,2,3)
addEdge(L,3,4)
printGraph(L)

C:\test>py test.py
0 [1, 4]
1 [0, 2, 3, 4]
2 [1, 3]
3 [1, 2, 4]
4 [0, 1, 3]
```

Stack ---> display LIFO
Queue ---> display FIFO
LL ------> display manually
Tree ----> inorder, preorder, postorder & level order
Graphs --> BFS and DFS

Bredth First Search Algorithm

We have to visit each node exactly once.

1. start from source vertex.
2. try to fetch adjancent nodes for node.
3. visit all adjacent nodes
4. repeat this process for each node present in grpah

```
from collections import deque

def addEdge(L,u,v):
	L[u].append(v)
	L[v].append(u)

def printGraph(L):
	for i in range(len(L)):
		print(i,L[i])

def bfs(L,source):
	visited = [False] * len(L)
	q = deque()
	q.append(source)
	visited[source] = True

	while q:
		source = q.popleft()
		print(source,end=' ')
		for i in L[source]:
			if visited[i] == False:
				q.append(i)
				visited[i] = True


'''
#Graph1
v = 4
L = [[] for i in range(v)]
```

```
addEdge(L,0,1)
addEdge(L,0,2)
addEdge(L,1,2)
addEdge(L,1,3)
addEdge(L,2,3)
printGraph(L)
print("BFS of Graph:")
bfs(L,0) #0 1 2 3
'''

#Graph2
v = 6
L = [[] for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,2)
addEdge(L,0,5)
addEdge(L,1,3)
addEdge(L,2,4)
addEdge(L,3,5)
addEdge(L,4,5)
printGraph(L)
print("BFS of Graph:")
bfs(L,0)

C:\test>py test.py
0 [1, 2, 5]
1 [0, 3]
2 [0, 4]
3 [1, 5]
4 [2, 5]
5 [0, 3, 4]
BFS of Graph:
0 1 2 5 3 4
```

Depth First Search Algorithm

We have to visit each node exactly once.

1. start from source vertex.
2. try to fetch its first adjancent nodes for node.
3. visit till depth by using recursion
4. repeat this process for each node present in grpah

```
from collections import deque

def addEdge(L,u,v):
      L[u].append(v)
      L[v].append(u)

def printGraph(L):
      for i in range(len(L)):
            print(i,L[i])

def bfs(L,source):
      visited = [False] * len(L)
      q = deque()
      q.append(source)
      visited[source] = True

      while q:
            source = q.popleft()
            print(source,end=' ')
```

```
            for i in L[source]:
                  if visited[i] == False:
                        q.append(i)
                        visited[i] = True

def dfs(L,source):
      visited = [False] * len(L)
      dfsUtil(L,source,visited)


def dfsUtil(L,source,visited):
      visited[source] = True
      print(source,end=' ')
      for i in L[source]:
            if visited[i] == False:
                  dfsUtil(L,i,visited)

'''
#Graph1
v = 4
L = [[] for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,2)
addEdge(L,1,2)
addEdge(L,1,3)
addEdge(L,2,3)
printGraph(L)
print("BFS of Graph:")
bfs(L,0)
print()
print("DFS of Graph:")
dfs(L,0)

'''
#Graph2
v = 6
L = [[] for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,2)
addEdge(L,0,5)
addEdge(L,1,3)
addEdge(L,2,4)
addEdge(L,3,5)
addEdge(L,4,5)
printGraph(L)
print("BFS of Graph:")
bfs(L,0) #0 1 2 5 3 4
print()
print("DFS of Graph:")
dfs(L,0) #0 1 3 5 4 2
```

Intro Graph
Representation of Graph
Matrix
List
BFS
DFS

Has Path?

```
from collections import deque

def addEdge(L,u,v):
      L[u].append(v)
```

```
      L[v].append(u)

def printGraph(L):
      for i in range(len(L)):
            print(i,L[i])

def bfs(L,source):
      visited = [False] * len(L)
      q = deque()
      q.append(source)
      visited[source] = True

      while q:
            source = q.popleft()
            print(source,end=' ')
            for i in L[source]:
                  if visited[i] == False:
                        q.append(i)
                        visited[i] = True

def dfs(L,source):
      visited = [False] * len(L)
      dfsUtil(L,source,visited)


def dfsUtil(L,source,visited):
      visited[source] = True
      print(source,end=' ')
      for i in L[source]:
            if visited[i] == False:
                  dfsUtil(L,i,visited)

def dfsUtil(L,curr,visited):
      visited[curr]=True
      LL = L[curr]
      for i in LL:
            if visited[i]==False:
                  dfsUtil(L,i,visited)


def hasPath(L,source,dest):
      visited = [False] * len(L)
      dfsUtil(L,source,visited)
      return visited[dest]

v = 6
L = [[] for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,3)
addEdge(L,1,2)
addEdge(L,3,2)
addEdge(L,2,5)
printGraph(L)
print(hasPath(L,0,5)) #True
print(hasPath(L,0,4)) #False
```

Count Paths

```
from collections import deque

def addEdge(L,u,v):
      L[u].append(v)
      L[v].append(u)
```

```
def printGraph(L):
	for i in range(len(L)):
		print(i,L[i])

def bfs(L,source):
	visited = [False] * len(L)
	q = deque()
	q.append(source)
	visited[source] = True

	while q:
		source = q.popleft()
		print(source,end=' ')
		for i in L[source]:
			if visited[i] == False:
				q.append(i)
				visited[i] = True

def dfs(L,source):
	visited = [False] * len(L)
	dfsUtil(L,source,visited)


def dfsUtil(L,source,visited):
	visited[source] = True
	print(source,end=' ')
	for i in L[source]:
		if visited[i] == False:
			dfsUtil(L,i,visited)

def dfsUtil(L,curr,visited):
	visited[curr]=True
	LL = L[curr]
	for i in LL:
		if visited[i]==False:
			dfsUtil(L,i,visited)


def hasPath(L,source,dest):
	visited = [False] * len(L)
	dfsUtil(L,source,visited)
	return visited[dest]

def countPaths(L,source,dest):
	visited = [False]*len(L)
	return countPathsUtil(L,visited,source,dest)

def countPathsUtil(L,visited,source,dest):
	if source==dest:
		return 1
	c=0
	visited[source] = True
	LL = L[source]
	for i in LL:
		if visited[i]==False:
			c=c+countPathsUtil(L,visited,i,dest)
	visited[source]=False
	return c




v = 6
L = [[] for i in range(v)]
```

```
addEdge(L,0,1)
addEdge(L,0,2)
addEdge(L,0,5)
addEdge(L,1,3)
#addEdge(L,2,4)
addEdge(L,3,5)
addEdge(L,4,5)
printGraph(L)
print(countPaths(L,0,5)) #3
print(countPaths(L,0,4)) #3
print(countPaths(L,0,2)) #3
```

Print All Path

```
from collections import deque

def addEdge(L,u,v):
      L[u].append(v)
      L[v].append(u)

def printGraph(L):
      for i in range(len(L)):
            print(i,L[i])

def bfs(L,source):
      visited = [False] * len(L)
      q = deque()
      q.append(source)
      visited[source] = True

      while q:
            source = q.popleft()
            print(source,end=' ')
            for i in L[source]:
                  if visited[i] == False:
                        q.append(i)
                        visited[i] = True

def dfs(L,source):
      visited = [False] * len(L)
      dfsUtil(L,source,visited)


def dfsUtil(L,source,visited):
      visited[source] = True
      print(source,end=' ')
      for i in L[source]:
            if visited[i] == False:
                  dfsUtil(L,i,visited)

def dfsUtil(L,curr,visited):
      visited[curr]=True
      LL = L[curr]
      for i in LL:
            if visited[i]==False:
                  dfsUtil(L,i,visited)


def hasPath(L,source,dest):
      visited = [False] * len(L)
      dfsUtil(L,source,visited)
      return visited[dest]

def countPaths(L,source,dest):
```

```
    visited = [False]*len(L)
    return countPathsUtil(L,visited,source,dest)

def countPathsUtil(L,visited,source,dest):
    if source==dest:
        return 1
    c=0
    visited[source] = True
    LL = L[source]
    for i in LL:
        if visited[i]==False:
            c=c+countPathsUtil(L,visited,i,dest)
    visited[source]=False
    return c

def printAllPaths(L,source,dest):
    visited = [False]*len(L)
    path = []
    printAllPathsUtil(L,visited,source,dest,path)

def printAllPathsUtil(L,visited,source,dest,path):
    path.append(source)
    if source==dest:
        print(path)
        path.pop()
        return
    visited[source] = True
    LL = L[source]
    for i in LL:
        if visited[i]==False:
            printAllPathsUtil(L,visited,i,dest,path)
    visited[source]=False
    path.pop()




v = 6
L = [[] for i in range(v)]
addEdge(L,0,1)
addEdge(L,0,2)
addEdge(L,0,5)
addEdge(L,1,3)
#addEdge(L,2,4)
addEdge(L,3,5)
addEdge(L,4,5)
printGraph(L)
printAllPaths(L,0,5)
```

Shorest Path
Minimum Cost Spanning Tree