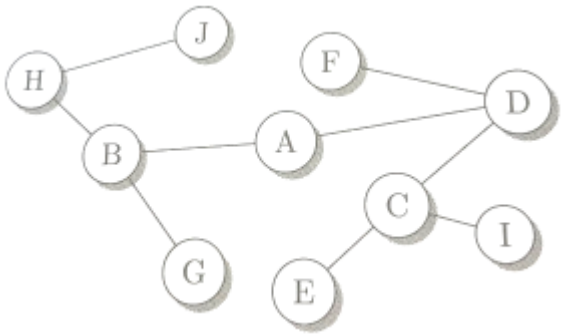
TD 15 : Parcours des graphes

Exercice 1 : Parcours en largeur et en profondeur

On considère le graphe ci-dessous :

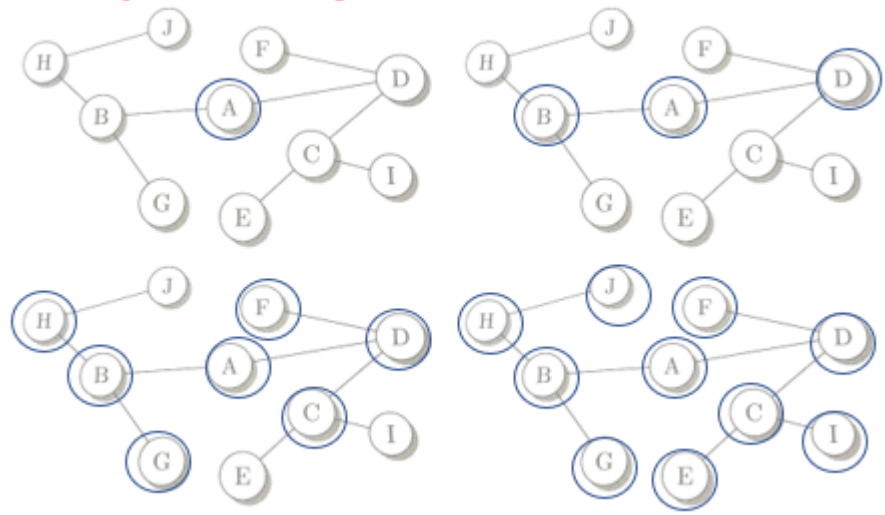


Expliciter comment on parcourt ce graphe à partir du somment initial A avec:

- un parcours en largeur (utilisant des files)
- un parcours en profondeur (utilisant des piles). On supposera que les listes d'adjacences associées à ce graphe classent les sommets dans l'ordre alphabétique :

In [5]:
```
g={"A":["B","D"],
   "B":["G","H"],
   "C":["D","E","I"],
   "D":["A","C","F"],
   "E":["C"],
   "F":["D"],
   "G":["B"],
   "H":["B","J"],
   "I":["C"],
   "J":["H"]}
```

Avec un parcours en largeur :



La file attente évolue de la manière suivante :

$[A], [B, D], [D, G, H], [G, H, C, F], [H, C, F], [C, F, J], [F, J, E, I], [J, E, I], [E, I], [I], []$

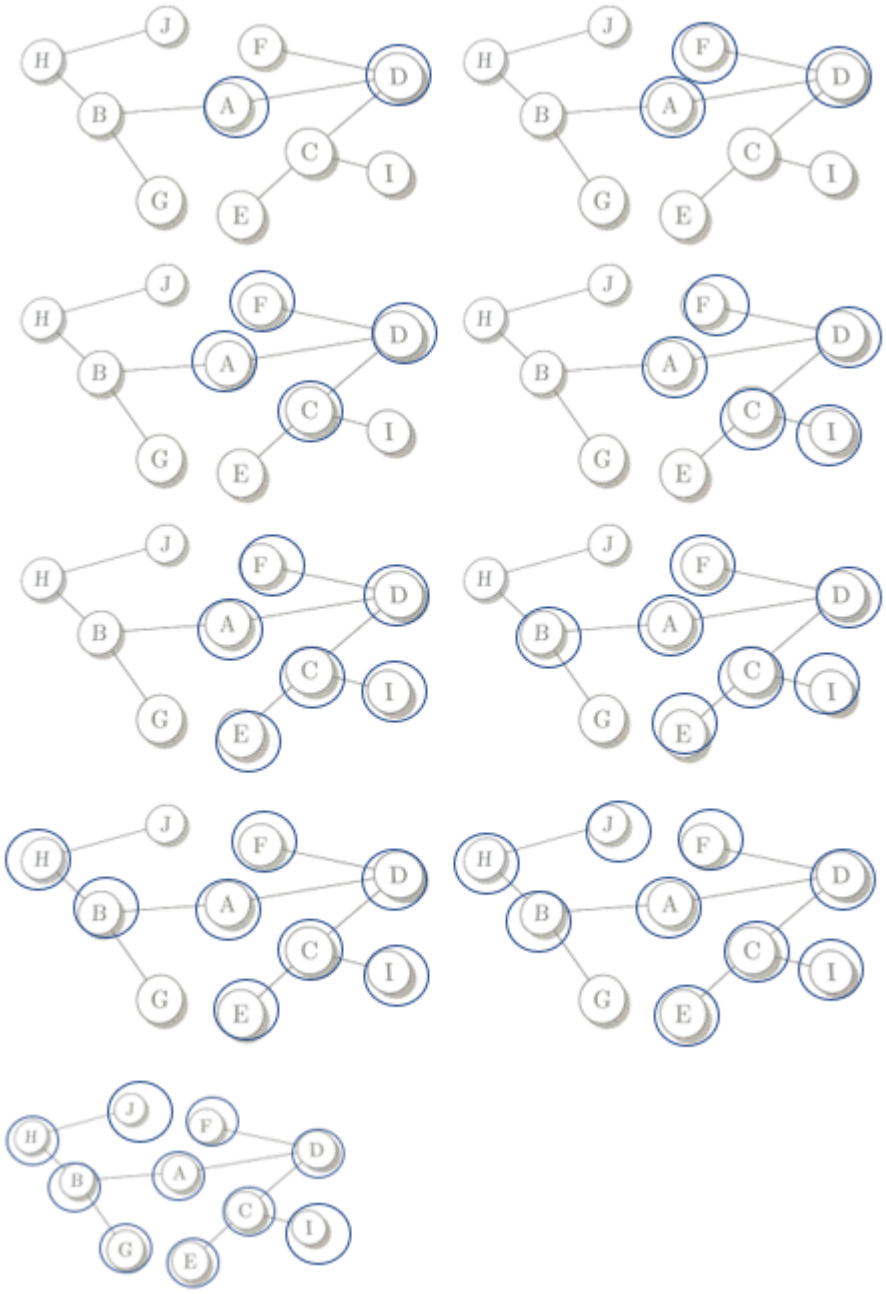
Et visite est :

$[A, B, D, G, H, C, F, J, E, I]$

In [6]:
```
from collections import deque
def pl_iteratif(g,sommet):
    visite=[]
    attente=deque([sommet])
    while len(attente)!=0:
        sommet=attente.popleft()
        if sommet not in visite :
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite :
                    attente.append(s)
    return visite
pl_iteratif(g,"A")
```

Out[6]:
```
['A', 'B', 'D', 'G', 'H', 'C', 'F', 'J', 'E', 'I']
```

Avec un parcours en profondeur :



La pile attente évolue de la manière suivante :

$[A], [B, D], [B, C, F], [B, C], [B, E, I], [B, E], [B], [G, H], [G, J], [G], []$

Et visite est :

$[A, D, F, C, I, E, B, H, J, G]$

In [8]:
```python
from collections import deque
def pp_iteratif(g,sommet):
    """g : dictionnaire des listes d'adjacence
    sommet : sommet de départ"""
    visite=[]#aucun sommet encore rencontré
    attente=deque([sommet])#initialisation
    while len(attente) !=0 :
        print(attente,visite)
        sommet=attente.pop()#le sommet visité est le dernier de la
        if sommet not in visite :
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite :
                    attente.append(s)
    return visite
```

In [9]:
```python
pp_iteratif(g,"A")
```

```
deque(['A']) []
deque(['B', 'D']) ['A']
deque(['B', 'C', 'F']) ['A', 'D']
deque(['B', 'C']) ['A', 'D', 'F']
deque(['B', 'E', 'I']) ['A', 'D', 'F', 'C']
deque(['B', 'E']) ['A', 'D', 'F', 'C', 'I']
deque(['B']) ['A', 'D', 'F', 'C', 'I', 'E']
deque(['G', 'H']) ['A', 'D', 'F', 'C', 'I', 'E', 'B']
deque(['G', 'J']) ['A', 'D', 'F', 'C', 'I', 'E', 'B', 'H']
deque(['G']) ['A', 'D', 'F', 'C', 'I', 'E', 'B', 'H', 'J']
```

Out[9]:
```
['A', 'D', 'F', 'C', 'I', 'E', 'B', 'H', 'J', 'G']
```

Exercice 2 : Parcours en profondeur

1) Proposer une fonction pp_it(g,s) réalisant le parcours en profondeur itératif d'un graphe associé à un dictionnaire g des listes d'adjacence à partir d'un sommet s, on utilise une pile pour placer les sommets en attente. Cette fonction renvoie la liste de tous les sommets accessibles depuis s.

2) Donner l'état de la pile attente et de la liste visite à chaque itération en partant du sommet A.

3) Reprendre les mêmes questions avec une programmation récursive. La fonction pp_rec(g,attente,visite) prend en argument le dictionnaire des listes d'adjacence g, une pile attente (contenant au premier appel le sommet s de départ) et une liste visite (initialement vide).

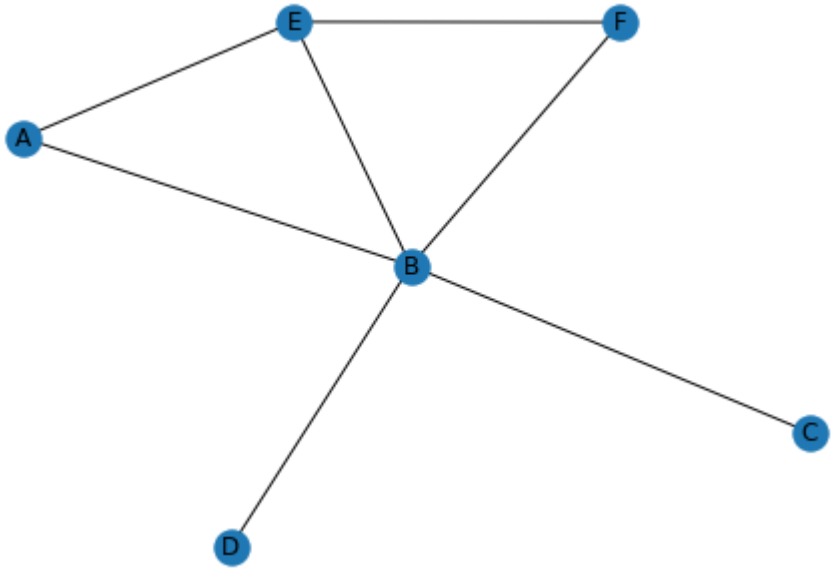
On considère la graphe suivant :

In [10]:

```
g={"A":["B","E"],
   "B":["A","C","D","E","F"],
   "C":["B"],
   "D":["B"],
   "E":["A","B","F"],
   "F":["B","E"]}
import networkx as nx
import matplotlib.pyplot as plt

G=nx.Graph(g)

# Tracé
plt.figure()
nx.draw(G, with_labels=True)
plt.show()
```



In [11]:
```
def pp_it(g,sommet):
    visite=[]
    attente=deque([sommet])
    while len(attente)!=0:
        print(attente)
        print(visite)
        sommet=attente.pop()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite:
                    attente.append(s)
    return visite
pp_it(g,"A")
```

```
deque(['A'])
[]
deque(['B', 'E'])
['A']
deque(['B', 'B', 'F'])
['A', 'E']
deque(['B', 'B', 'B'])
['A', 'E', 'F']
deque(['B', 'B', 'C', 'D'])
['A', 'E', 'F', 'B']
deque(['B', 'B', 'C'])
['A', 'E', 'F', 'B', 'D']
deque(['B', 'B'])
['A', 'E', 'F', 'B', 'D', 'C']
deque(['B'])
['A', 'E', 'F', 'B', 'D', 'C']
```

Out[11]:
```
['A', 'E', 'F', 'B', 'D', 'C']
```

On a donc visite = $[A], [A, E], [A, E, F], [A, E, F, B], [A, E, F, B, D], [A, E, F, B, D, C]$

attente = $[A][B, E], [B, B, F], [B, B, B], [B, B, C, D], [B, B, C], [B, B], [B], []$

In [21]:
```
def pp_rec(g,attente,visite):
    if len(attente)==0:
        return visite
    else :
        sommet=attente.pop()
        if sommet not in visite :
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite :
                    attente.append(s)
        return pp_rec(g,attente,visite)
```

In [22]:
```
pp_rec(g,deque(["A"]),[])
```

Out[22]:
```
['A', 'E', 'F', 'B', 'D', 'C']
```

Le parcours du graphe est tout à fait analogue. A chaque appel, on stocke les prochains voisins et on dépile les même sommets

Exercice 3 : Parcours en largeur

1) Proposer une fonction pl_it(g,s) réalisant le parcours en largeur itératif d'un graphe associé à un dictionnaire g des listes d'adjacence à partir d'un sommet s, on utilise une file pour placer les sommets en attente. Cette fonction renvoie la liste de tous les sommets accessibles depuis s.

2) Donner l'état de la file attente et de la liste visite à chaque itération.

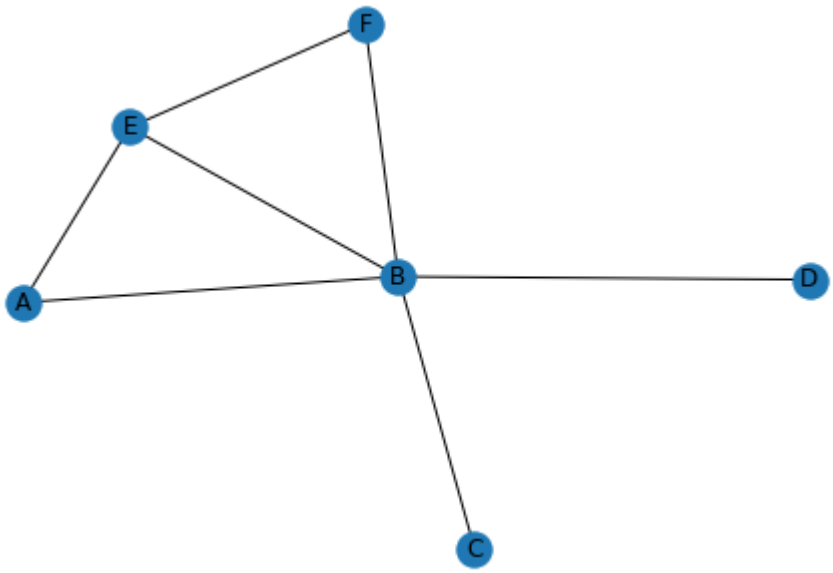
3) Reprendre les mêmes questions avec une programmation récursive. La fonction pl_rec(g,attente,visite) prend en argument le dictionnaire des listes d'adjacence g, une file attente (contenant au premier appel le sommet s de départ) et une liste visite (initialement vide).

On considère la graphe suivant :

In [23]:
```
g={"A":["B","E"],
   "B":["A","C","D","E","F"],
   "C":["B"],
   "D":["B"],
   "E":["A","B","F"],
   "F":["B","E"]}
import networkx as nx
import matplotlib.pyplot as plt

G=nx.Graph(g)

# Tracé
plt.figure()
nx.draw(G, with_labels=True)
plt.show()
```



In [29]:
```
def pl_it(g,sommet):
    visite=[]
    attente=deque([sommet])
    while len(attente)!=0:
        print(attente)
        sommet=attente.popleft()
        if sommet not in visite:
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite :
                    attente.append(s)
    return visite
```

In [30]:
```
pl_it(g,"A")
```

```
deque(['A'])
deque(['B', 'E'])
deque(['E', 'C', 'D', 'E', 'F'])
deque(['C', 'D', 'E', 'F', 'F'])
deque(['D', 'E', 'F', 'F'])
deque(['E', 'F', 'F'])
deque(['F', 'F'])
deque(['F'])
```

Out[30]:
```
['A', 'B', 'E', 'C', 'D', 'F']
```

On a donc visite = $[A], [A, B], [A, B, E], [A, B, E, C], [A, B, E, C, D], [A, B, E, C, D, F]$

attente = $[A][B, E], [E, C, D, E, F], [C, D, E, F, F], [D, E, F, F], [E, F, F], [F, F], [F], [$

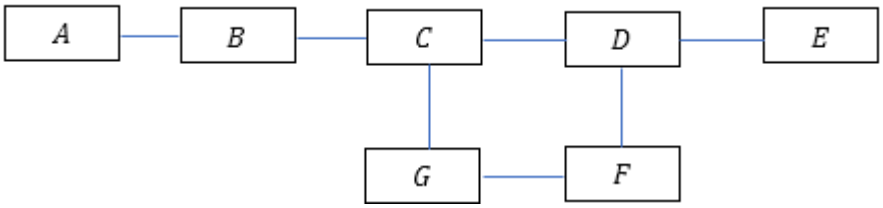
In [33]:
```
def pl_rec(g,attente,visite):
    if len(attente)==0:
        return visite
    else :
        sommet=attente.popleft()
        if sommet not in visite :
            visite.append(sommet)
            for s in g[sommet]:
                if s not in visite :
                    attente.append(s)
        return pl_rec(g,attente,visite)
```

In [34]:
```
pl_rec(g,deque(["A"]),[])
```

Out[34]:
```
['A', 'B', 'E', 'C', 'D', 'F']
```

Exercice 4 : Détermination de la présence d'un cycle dans un graphe

Une molécule peut être décrite comme un graphe, les atomes jouant le rôle de sommets et les liaisons jouant le rôle d'arrêtes. Certaines molécules présentent des cycles d'atomes. On souhaite étudier un algorithme permettant de repérer la présence de cycles dans un graphe non orienté et non pondéré. Le graphe étudié est :



1) Créer « manuellement » sur python le dictionnaire des listes d'adjacence du graphe ci-dessus.

On donne le programme ci-dessous :

In [35]:
```
def cycle(g,sommet):
    """g est un dictionnaire des listes d'adjacence du graphe
    sommet est le sommet de départ"""
    niveaux={s:None for s in g}
    niveaux[sommet]=0
    file=deque()
    file.append(sommet)
    while len(file)>0:
        sommet=file.popleft()
        for s in g[sommet]:
            if niveaux[s]==None:
                niveaux[s]=niveaux[sommet]+1
                file.append(s)
            elif niveaux[s]>=niveaux[sommet]:
                return True
    return False

```

2) En utilisant le graphe ci-dessus, expliquer le principe de détection d'un cycle dans un graphe du programme proposé.

3) Proposer un algorithme de recherche d'un cycle basé sur un parcours en profondeur.

In [59]:

```python
from collections import deque
g={"A":["B"],
   "B":["A","C"],
   "C":["B","D","G"],
   "D":["C","E","F"],
   "E":["D"],
   "F":["D","G"],
   "G":["C","F"]}

def cycle(g,sommet):
    """g est un dictionnaire des listes d'adjacence du graphe
    sommet est le sommet de départ"""
    niveaux={s:None for s in g}# on a la valeur None affecté à chaq
    niveaux[sommet]=0#le 1e sommet est inidqué comme visité avec 0
    file=deque()#file vide  file jous le rôle de file d'attente
    file.append(sommet)
    while len(file)>0:
        sommet=file.popleft()#=> parcours en largeur,
        for s in g[sommet]:
            if niveaux[s]==None:
                niveaux[s]=niveaux[sommet]+1#on ajoute +1 au niveau
                file.append(s)# on ajoute ces sommets pour les test
            elif niveaux[s]>=niveaux[sommet]:# si le sommet voisin
                return True
    return False
cycle(g,"A")
```

Out[59]: True

niveau = {A : None, B:None....}

file =[A], niveau {A:0, B:1, C: None....}

file =[B], niveau {A:0, B:1, C: 2, D:None....}

file =[C], niveau {A:0, B:1, C: 2, D:3, G:3....}

file =[D,G], niveau {A:0, B:1, C: 2, D:3, G:3, E=4,F=4}#tous les sommets sont vus!

file =[G,E,F], niveau {A:0, B:1, C: 2, D:3, G:3, E=4,F=4}#on compare niveau[G]=3 à niveau[C]=2 => cycle !

In [60]:
```python
#pour un parcours en profondeur
def cycle_profondeur(g,sommet):
    """g est un dictionnaire des listes d'adjacence du graphe
    sommet est le sommet de départ"""
    niveaux={s:None for s in g}
    niveaux[sommet]=0
    pile=deque()
    pile.append(sommet)
    while len(pile)>0:
        sommet=pile.pop()
        for s in g[sommet]:
            if niveaux[s]==None:
                niveaux[s]=niveaux[sommet]+1
                pile.append(s)
            elif niveaux[s]>=niveaux[sommet]:
                return True
    return False
cycle_profondeur(g,"A")
```

Out[60]: True

niveau = {A : None, B:None....}

file =[A], niveau {A:0, B:1, C: None....}

file =[B], niveau {A:0, B:1, C: 2, D:None....}

file =[C], niveau {A:0, B:1, C: 2, D:3, G:3....}

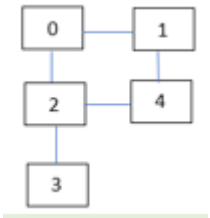
file =[D,G], niveau {A:0, B:1, C: 2, D:3, G:3, F:4}#tous les sommets sont vus!

file =[G,F], niveau {A:0, B:1, C: 2, D:3, G:3, F=4}

file =[G], niveau {A:0, B:1, C: 2, D:3, G:3, F=4}#niveau[F]>niveau[G]

Exercice 5 : Parcours en profondeur d'une matrice d'adjacence avec numpy

On considère le graphe non orienté et non pondéré ci-contre pour lequel les sommets sont notés 0,1,2,3,4. A l'aide du module numpy on créer la matrice d'adjacence :



In [13]:
```python
"""on déclare les sommets"""
S = [k for k in range(0,5)]
"""on déclare les arêtes"""
A= [[0,1],[0,2],[1,4],[2,4],[2,3]]

```

1) Obtenir l'expression de la matrice M des listes d'adjacence à l'aide d'une fonction prenant pour argument S et A et utilisant les tableaux numpy.

2) En vous inspirant des algorithmes du cours, proposer un programme permettant d'effectuer :

- Un parcours en profondeur (récursif ou itératif)
- Un parcours en largeur (récursif ou itératif)

Ces programmes permettront d'avoir la liste des sommets accessible depuis un sommet de départ donné.

3) Proposer un programme permettant de savoir si un graphe est connexe.

In [14]:
```
import numpy as np
def matrice(S,A):
    n=len(S)
    M=np.zeros((n,n))
    for i in range(len(A)) :
        deb,fin=A[i][0],A[i][1]
        M[deb][fin]=M[fin][deb]=1
    return M
matrice (S,A)
```

Out[14]:
```
array([[0., 1., 1., 0., 0.],
       [1., 0., 0., 0., 1.],
       [1., 0., 0., 1., 1.],
       [0., 0., 1., 0., 0.],
       [0., 1., 1., 0., 0.]])
```

In [16]:

```
m=matrice(S,A)
from collections import deque
def profondeur_it(m,depart):
    visite=[]
    attente=deque([])
    attente.append(depart)
    while len(attente)!=0:
        sommet=attente.pop()
        if sommet not in visite:
            visite.append(sommet)
            for voisin in range(len(m)):
                if m[sommet][voisin]==1 and voisin not in visite :
                    attente.append(voisin)
    return visite
print(profondeur_it(m,0))

def largeur_it(m,depart):
    visite=[]
    attente=deque([])
    attente.append(depart)
    while len(attente)!=0:
        sommet=attente.popleft()
        if sommet not in visite:
            visite.append(sommet)
            for voisin in range(len(m)):
                if m[sommet][voisin]==1 and voisin not in visite :
                    attente.append(voisin)
    return visite
print(largeur_it(m,0))


def profondeur_rec(m,visite,attente):
    if len(attente)==0:
        return visite
    else :
        sommet=attente.pop()
        if sommet not in visite :
            visite.append(sommet)
            for voisin in range(len(m)):
                if m[sommet][voisin]==1 and voisin not in visite :
                    attente.append(voisin)
        return profondeur_rec(m,visite,attente)
print(profondeur_rec(m,[],deque([0])))

def largeur_rec(m,visite,attente):
    if len(attente)==0:
        return visite
    else :
        sommet=attente.popleft()
        if sommet not in visite :
            visite.append(sommet)
            for voisin in range(len(m)):
                if m[sommet][voisin]==1 and voisin not in visite :
                    attente.append(voisin)
        return largeur_rec(m,visite,attente)
print(largeur_rec(m,[],deque([0])))

def connexe(m):
    n=len(m)
    for i in range(n):
        if len(largeur_it(m,i))!=n:
```

```
            return False
    return True

connexe(m)
```

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

Out[16]: True

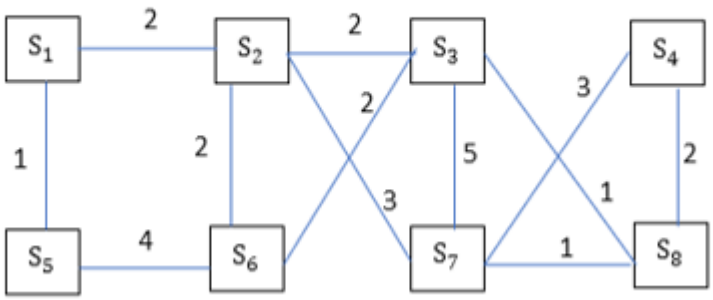
Exercice 6 : Principe de l'algorithme de Dijkstra

Il s'agit d'un algorithme qui fait le choix localement optimal du plus court chemin pour atteindre chaque sommet d'un graphe pondéré depuis un sommet de départ S_1 . A la fin, ce programme fournit les meilleurs chemins pour relier n'importe quel sommet S_i au sommet de départ S_1 :

- On part donc du sommet initial S_1 et on analyse ses premiers voisins en mesurant leur distance depuis S_1
- Pour chaque voisin, on crée un couple (sommets, distance) que l'on met dans une file de priorité et qui place en premier le sommet S_i le plus proche de $S1$
- Ce couple est alors choisi (c'est le chemin optimal $S_1 S_i$, les autres chemins menant à S_i sont plus longs et ignorés)
- On ajoute à la file, les sommets atteignables depuis S_i. Les distances sont calculées depuis $S1$
- Et on défile la file de priorité en retenant le sommet suivant le plus proche de S_1

Donner l'évolution de la file de priorité lors du parcours du graphe ci-dessous :

$$(S_1,0),(S_2,\infty),(S_3,\infty),(S_4,\infty),(S_5,\infty),(S_6,\infty),(S_7,\infty),(S_8,\infty)$$



file = [(S1,0)] # S1 est choisit

file = [(S1S5,1),(S1S2,2)] # S1S5 est choisit, distance 1

file = [(S1S2,2),(S1S5S6,5)] # S1S2 est choisit, distance

file = [(S1S2S6,4),(S1S2S3,4),(S1S2S7,5),(S1S5S6,5)] #on choisit S1S2S6, distance 4

file = [(S1S2S3,4),(S1S2S7,5),(S1S2S6S3,6)]#on choisit S1S2S3, distance 4

file = [(S1S2S7,5),(S1S2S3S8,5),(S1S2S3S7,7]#onchoisit S1S2S7, distance 5

file = [(S1S2S3S8,5),(S1S2S7S8,6),(S1S2S7S4,8)]#onchoisit S1S2S3S8S8, distance 5

file = [(S1S2S3S8S4,7)(S1S2S7S4,8)]#onchoisit S1S2S3S8S4, distance 7

Exercice 7 : Implémentation de l'algorithme de Dijkstra

L'implémentation de la file de priorité se faite avec la fonction *queue* de la librairie PriorityQueue

In [ ]:
```
from queue import PriorityQueue
#on rappelle les éléments suivants concernant les files de priorité
file_priorite=PriorityQueue()#initialisation
file_priorite.put((valeur,nom))#remplissage avec le 1e argument qui
print(file_priorite.empty())#renvoie True si vide
valeur,nom=file_priorite.get()#par défaut renvoie le couple associé
```

In [2]:
```
from queue import PriorityQueue
def dijkstra(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet de dép
    en prenant une file de priorité, on assure  le schéma de dijkst
    visite={}#initialisation d'un dictionnaire cle = sommet, valeur
    attente=PriorityQueue()#file de priorité
    attente.put((0,sommet))#on ajoute (distance,sommet)
    while not attente.empty() :#si file non vide
        valeur,sommet=attente.get()#on enlève le couple avec la plu
        if sommet not in visite :
            visite[sommet]=valeur
            for s,v in g[sommet] :
                if s not in visite :
                    attente.put((valeur+v,s))#on ajoute les « infos
    return visite
```

On reprend l'exemple de l'exercice précédent :

In [7]:
```
g={"S1":[("S2",2),("S5",1)],
   "S2":[("S1",2),("S3",2),("S6",2),("S7",3)],
   "S3":[("S2",2),("S6",2),("S7",5),("S8",1)],
   "S4":[("S7",3),("S8",2)],
   "S5":[("S1",1),("S6",4)],
   "S6":[("S2",2),("S5",4),("S3",2)],
   "S7":[("S2",3),("S3",5),("S4",3),("S8",1)],
   "S8":[("S3",1),("S4",2),("S7",1)]}
```

1) Prévoir l'évolution du dictionnaire visite à chaque itération après l'appel dijkstra(g,"S1")

visite ={"S1":0}

visite ={"S1":0,"S5":1}

visite ={"S1":0,"S5":1,"S2":2}

visite ={"S1":0,"S5":1,"S2":2,"S6":4}

visite ={"S1":0,"S5":1,"S2":2,"S6":4,""S3":4}

visite ={"S1":0,"S5":1,"S2":2,"S6":4,""S3":4,"S7":5}

visite ={"S1":0,"S5":1,"S2":2,"S6":4,""S3":4,"S7":5,"S8":5}

visite ={"S1":0,"S5":1,"S2":2,"S6":4,""S3":4,"S7":5,"S8":5,"S4":7}

2) Modifier le programme précédent afin d'obtenir, pour chaque sommet, la longueur ainsi que les sommets par lesquels il faut passer pour assurer un chemin optimal

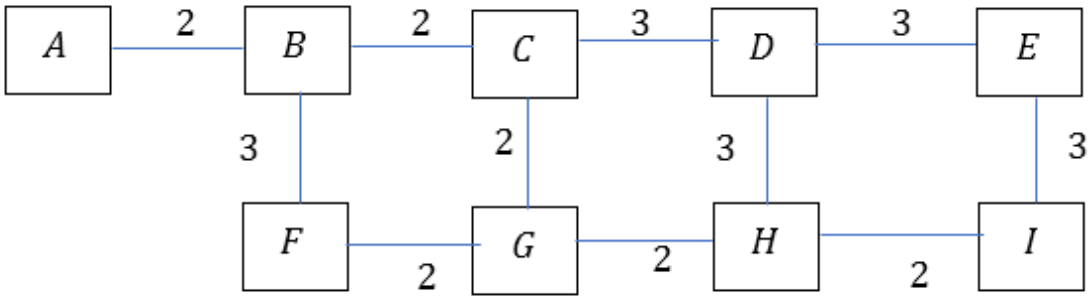
In [10]:
```
def dijkstra(g,sommet):
    """g est le dictionnaire des listes d'adjacence
    sommet est le sommet à visiter (initialisé sur le sommet de dép
    en prenant une file de priorité, on assure  le schéma de dijkst
    visite={}#initialisation d'un dictionnaire cle = sommet, valeur
    attente=PriorityQueue()#file de priorité
    chemin=sommet
    attente.put((0,sommet,chemin))#on ajoute (distance,sommet,chemi
    while not attente.empty() :#si file non vide
        valeur,sommet,chemin=attente.get()
        if sommet not in visite :
            visite[sommet]=(chemin,valeur)
            for s,v in g[sommet] :
                if s not in visite :
                    attente.put((valeur+v,s,chemin+s))#on ajoute le
    return visite
dijkstra(g,"S1")
```

Out[10]:
```
{'S1': ('S1', 0),
 'S5': ('S1S5', 1),
 'S2': ('S1S2', 2),
 'S3': ('S1S2S3', 4),
 'S6': ('S1S2S6', 4),
 'S7': ('S1S2S7', 5),
 'S8': ('S1S2S3S8', 5),
 'S4': ('S1S2S3S8S4', 7)}
```

Exercice 8 : Métro c'est trop

Vous avez à disposition des données sur un métro :

- Les sommets, représentés ici par des lettres, correspondent aux noms de stations
- Les valeurs sur les arrêtes correspondent aux temps de trajet nécessaire entre les sommets associés



On donne le dictionnaire associé à ce graphe

In [14]:
```
from queue import PriorityQueue
g={"A":[("B",2)],
   "B":[("A",2),("C",2),("F",3)],
   "C":[("B",2),("D",3),("G",2)],
   "D":[("C",3),("E",3),("H",3)],
   "E":[("D",3),("I",3)],
   "F":[("B",3),("G",2)],
   "G":[("C",2),("F",2),("H",2)],
   "H":[("G",2),("I",2),("D",3)],
   "I":[("H",2),("E",2)]}
```

Adapter le programme de Dijkstra (qui calcule depuis A le meilleur parcours pour toutes les stations) de l'exercice précédent de manière à pouvoir obtenir le parcours à suivre entre une station de départ dep et une station d'arrivée der. Cette fonction prend en argument g,dep,der

In [31]:
```
from collections import deque
from queue import PriorityQueue

def metro(g,depart):
    visite={}
    attente=PriorityQueue()
    chemin=depart#initialisation
    station=depart
    temps=0
    attente.put((temps,station,chemin))
    while not attente.empty():
        temps,station,chemin=attente.get()
        if station not in visite:
            visite[station]=[chemin,temps]
            for s,t in g[station]:
                if s not in visite:
                    attente.put((temps + t, s, chemin+s))
    return visite
```

In [32]:
```
metro(g,"A")
```

Out[32]:
```
{'A': ['A', 0],
 'B': ['AB', 2],
 'C': ['ABC', 4],
 'F': ['ABF', 5],
 'G': ['ABCG', 6],
 'D': ['ABCD', 7],
 'H': ['ABCGH', 8],
 'E': ['ABCDE', 10],
 'I': ['ABCGHI', 10]}
```

In [38]:
```
from collections import deque
from queue import PriorityQueue

def metro2(g,depart,fin):
    visite={}
    attente=PriorityQueue()
    chemin=depart#initialisation
    station=depart
    temps=0
    attente.put((temps,station,chemin))
    while not attente.empty():
        temps,station,chemin=attente.get()
        if station not in visite:
            visite[station]=[chemin,temps]
            if station==fin:
                return visite[fin]
            for s,t in g[station]:
                if s not in visite:
                    attente.put((temps + t, s, chemin+s))
    return print("la station n'est pas accessible")
```

In [39]:
```
metro2(g,"A","I")
```

Out[39]:
```
['ABCGHI', 10]
```

In [ ]:
```

```