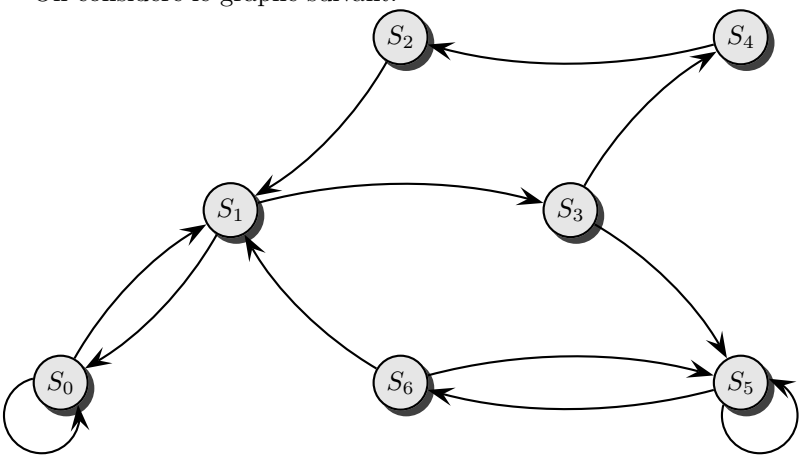
Exercice 1 *Graphe*

On considère le graphe suivant.



1. Donner la matrice d'adjacence M du graphe précédent.
2. Pour $i, j \in \{0, \ldots, 6\}$, que représente le coefficient $[M^2]_{i,j}$?
3. Le parcours en profondeur d'un graphe consiste à explorer tous les successeurs à partir d'un sommet initial, on explore aussi longtemps que possible sans jamais passer deux fois par le même sommet. Lorsqu'on est bloqué, on revient en arrière sommet par sommet jusqu'à trouver un voisin non visité. On explore alors dans cette direction jusqu'à être bloqué à nouveau etc.
   Effectuer ce parcours depuis le sommet S_0 en précisant toutes les étapes.

```python
# Variables globales
decouverts=[0] #liste des noeuds découverts
visites = [] #liste des noeuds visités
avisiter=[0] #liste des noeuds restants à visiter

def Profondeur(graph:list, courant:int)->list :
    '''fonction récursive locale ("visites" as global)
    Affiche dans stdout la liste des noeuds parcourus'''
    # On visite le noeud courant
    visites.append(courant)
    avisiter.remove(courant)
    for voisin in graph[courant] : # On visite les voisins du noeud courant
        if voisin not in decouverts :
            decouverts.append(voisin)  #mise à jour decouverts
        if voisin not in avisiter and voisin not in visites:
            avisiter.append(voisin) #mise à jour à visiter
        if voisin not in visites :
            Profondeur(graph, voisin) #si non déjà fait, on visite le noeud


# Test  : On indique le noeud source à gauche et les successeurs à suivre

G = [[0,0,1],[1,0,3],[2,1],[3,4,5],[4,2],[5,5,6],[6,1,5]]
Profondeur(G, 0)
print('\n','Visités en profondeur=',visites,'\n')
```

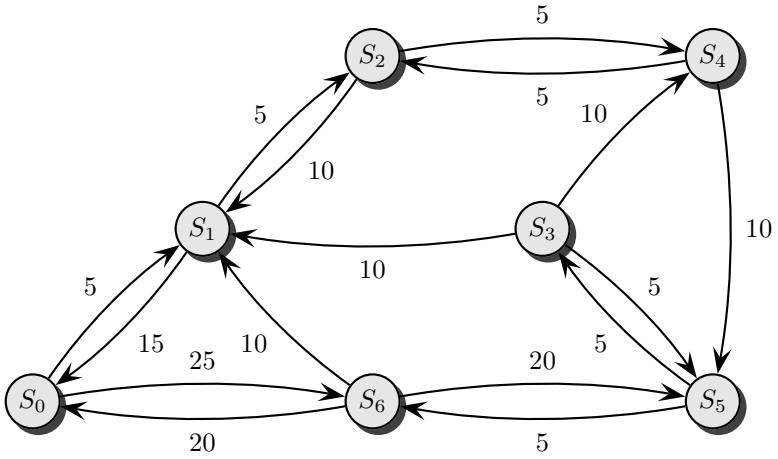
4. Le parcours en largeur a pour principe d'aller explorer les sommets du graphe par éloignement croissant (au sens du nombre d'arêtes minimum d'un chemin les reliant) avec un sommet de départ u.
   Effectuer ce parcours depuis le sommet S_0 en précisant toutes les étapes.

```
def largeur (graph:list, source:int)->list :
    '''# arguments   graphe à explorer et noeud origine.
     Renvoie la liste des coples
     (noeuds visité,plus court chemin pour l' atteindre)'''
    queue = [(source, [0])] # On initialise la queue
    visites = [source] # on met à jour les noeuds visités
    # On insère l'origine et le plus court chemin
    listeresultats=[]
    while queue :     # On explore
        (courant, chemin) = queue[0] # on extrait le noeud le plus ancien
        queue = queue[1:]
        listeresultats.append([courant,chemin])
        for voisin in graph[courant] :
            if voisin not in visites :
                # On ajoute les voisins non encore visités du noeud courant
                queue.append((voisin, chemin + [voisin]))
                visites.append(voisin)
    return(listeresultats)

# Test

G = [[0,0,1],[1,0,3],[2,1],[3,4,5],[4,2],[5,5,6],[6,1,5]]
print('\n','Visités en largeur=',largeur(G, 0),'\n')
```

5. On stocke en Python un graphe orienté sous la forme
   $G = [[sommet1, successeur1, ...successeurn1], [sommetm, successeur1, ...successeurnm]]$, ou chaque arc $(i- > j)$ correspond à mettrel'indice j parmi les successeurs de i
   Ecrire une fonction Python, `adjacence(G)` qui à un graphe G associe sa matrice d'adjacence.
6. On note N la matrice obtenue sur le graphe $G = [[0,0,1],[1,0,3],[2,1],[3,4,5],[4,2],[5,5,6],[6,1,5]]$
   Importer la bibliothèque `from numpy.linalg import matrix_power`
   Essayer les commandes `matrix_power(N,k)` pour k de 2 à 4
7. Que représentent les entiers présents dans la première ligne de N^2 ?
8. Tous les coefficients de la première ligne de N^4 sont supérieurs ou égaux à 1, que cela signifie-t-il en termes de connexité du graphe G ?

Exercice 2

On considère le graphe suivant dont les pondération représente un temps de trajet en minutes.



1. On donne le code suivant (copier-coller via le .pdf du cagier de textes des PC)

```
G=[[1,2,3],[4],[5],[],[2,5,7],[6],[7],[5]]

D=[[0,1,4,0.5,float("inf"),float("inf"),float("inf"),float("inf")],
 [float("inf"),0,float("inf"),float("inf"),10,float("inf"),float("inf"),float("inf")],
 [float("inf"),float("inf"),0,float("inf"),6,1,float("inf"),float("inf")],
 [float("inf"),float("inf"),float("inf"),0,float("inf"),float("inf"),float("inf"),float("inf")],
 [float("inf"),float("inf"),float("inf"),float("inf"),0,3,float("inf"),1],
 [float("inf"),float("inf"),float("inf"),float("inf"),float("inf"),0,3,float("inf")],
 [float("inf"),float("inf"),float("inf"),float("inf"),float("inf"),float("inf"),0,5]    ,
 [float("inf"),float("inf"),float("inf"),float("inf"),float("inf"),10,float("inf"),0]
    ]

print('Gliste adjacence'+'\n',G)

print('D matrice distances'+'\n',np.matrix(D))
```

Expliquer ce que représentent les objets G et D.

2. Executer l'algorithme de Dijkstra sur ce graphe pondéré pour trouver la distance minimale de S_0 à S_5.

```
#####################################################################
def minimum(couples:[(float,int)],sommets:[int])->int:
    """couples : liste de couples (distance,sommet)
    sommets : liste de sommets
    Renvoie, parmi la liste sommets, celui qui est présent
    dans la liste couples et dont la première
    composante est minimale"""
    d0=float("inf")
    for u in sommets:
        (d,s)=couples[u]
        if d<d0:
            smin=u
            d0=d
    return smin

#####################################################################
```

```
def dijkstra(g:[[int]],dist:[[float]], dep:int,arr:int)->(float,[int]):
    """g : liste d adjacence d un graphe
    dist : dist[i][j] distance de i à j
    dep : numéro sommet départ
    arr : numéro sommet arrivée
    Renvoie la distance minimale de dep à arr et la liste des sommets qui permettent de réaliser
    cette distance
    """
    n=len(g) #nombre de sommets
    a_visiter=list(range(n)) #Init points à visiter
    #Init : le seul point connu est celui de départ 0
    distances=[(float("inf"),0) for i in range(n)]
    distances[dep]=(0,dep)
    while a_visiter!=[]:
        #Point le plus proche à visiter parmi ceux calculés
        smin=minimum(distances, a_visiter)
        a_visiter.remove(smin)
        print('Dijkstra visite',smin)
        for j in g[smin]: #actualisation des voisins
            nd=distances[smin][0]+dist[smin][j]
            if nd<distances[j][0]:
                distances[j]=(nd,smin)
    #Reconstruction du chemin optimal
    chemin,point=[arr],arr
    while point!=dep:
        point=distances[point][1]
        chemin.append(point)
    chemin.reverse()
    return distances[arr][0],chemin

print('Dijkstra donne : (distance, liste sommets)',dijkstra(G,D, ...,...))
```

3. Dans l'algorithme A^*, on propose l'heuristique ci-dessous

```

#####################################################################
def h(p,q):    #heuristique
    #on cherche a visiter prioritairement
    #les sommets de numéro proche de l'arrivée q= arr avec  arr=5
    return (20*abs(p-q))

def minimumh(couples:[(float,int)],sommets:[int],h:callable)->int:
    """couples : liste de couples (distance,sommet)
    sommets : liste de sommets
    h : heuristique
    Renvoie, parmi la liste sommets, celui qui est présent
    dans la liste couples et dont la première
    composantedans couples et l'heuristique  est minimale"""
    d0=float("inf")
    for u in sommets:
        (d,s)=couples[u]
        cout=d+h(u)## on tient compte de l'heuristique
        if d<d0:
            smin=u
            d0=cout
    return smin
```

Exécuter `print([h(x,5) for x in range(7)])` et expliquer quels sommets sont les moins coûteux pour cette heuristique h.

4. Executer l'algorithme $A^\star$ sur ce graphe pondéré pour trouver une distance de S_0 à S_5. Est-elle minimale ?

```
#####################################################################
def astar(g:[[int]],dist:[[float]], dep:int,arr:int,h:callable)->(float,[int]):
    """g : liste d adjacence d un graphe
    dist : dist[i][j] distance de i à j
    dep : numéro sommet départ
    arr : numéro sommet arrivée
    Renvoie la distance minimale de dep à arr et la liste des sommets qui permettent de réaliser
    cette distance
    """
    n=len(g) #nombre de sommets
    a_visiter=list(range(n)) #Init points à visiter
    #Init : le seul point connu est celui de départ 0
    distances={s:(float("inf"),0) for s in range(n)}
    distances[dep]=(0,dep)
    smin=dep
    while smin!=arr:
        #Point le plus proche à visiter parmi ceux calculés
        smin=minimumh(distances, a_visiter,lambda x:h(x,arr))
        a_visiter.remove(smin)
        print('astar visite',smin)
        for j in g[smin]:  #actualisation des voisins
            nd=distances[smin][0]+dist[smin][j]
            if nd<distances[j][0]:
                distances[j]=(nd,smin)
    #Reconstruction du chemin optimal
    chemin,point=[arr],arr
    while point!=dep:
        point=distances[point][1]
        chemin.append(point)
    chemin.reverse()
    return distances[arr][0],chemin

print('Astar donne : (distance, liste sommets)',astar(...,..., 0,5, h))
```

5. Modifier l'heuristique h pour favoriser les sommets de petits numéros.

Exercice 3 *Véhicule Glouton*

Une voiture veut parcourir un trajet de 2000 km, son réservoir ne lui permet de faire que 300 km avant de refaire le plein, on donne une table de stations sur le trajet et la distance de celles ci au point de départ.

On donne la liste suivante, disponible en ligne via le cahier de textes d'info :

```
table = [['debut',0],['A',150],['B',350],['C',1500],['D',250],['E',450],['F',800],['G',600],
         ['H',1200],['I',1600],['J',1400],['K',650],['L',1500],['M',570],['N',1800],
         ['O',1100],['P',1750],['Fin',2000]]
```

1. Quelle est la stratégie gloutonne permettant d'avoir une réponse optimale au problème.
2. Si on est à une station, expliquer clairement les choses à faire pour trouver la station suivante.
3. Compléter la fonction suivante(table,indice_derniere) qui va donner la station située après la station d'indice indice_derniere la plus éloignée tout en restant à moins de 300 km de celle ci, si ce n'est pas possible la fonction doit retourner -1.

   Ici vous avez deux choix : vous pouvez considérer que la table est pré classée et dans ce cas il suffit d'examiner les indices après indice_dernier où examiner tous les indices de la table à chaque fois.

```
def suivante(table,indice_derniere) :
    indice_meilleure  = 0
    for indice in range(len(table)):
        if ..... :
            indice_meilleure = indice
    if indice_meilleure == 0 :
        return -1
    return indice_meilleure
```

4. En vous aidant de la fonction précédente faire une fonction qui va donner la liste des stations, obtenue grâce à l'algorithme glouton, où la voiture doit faire le plein. Il suffit d'appliquer la fonction précédente tant que la distance de la station à la fin est supérieure à 300 km et de stocker les stations successives dans une liste (vous pouvez utiliser append ou concaténer).

   On pourra compléter le code suivant :

```
def glouton(table):
    solution = []
    derniere_station = 0
    while ..... :
        solution = .....
        derniere_station = .....
    if ..... :
    #Si la dernière station n'est pas la fin c'est que la voiture ne peut pas aller à la fin
        return None
    solution = solution + [derniere_station]
    return [table[i][0] for i in solution]
```