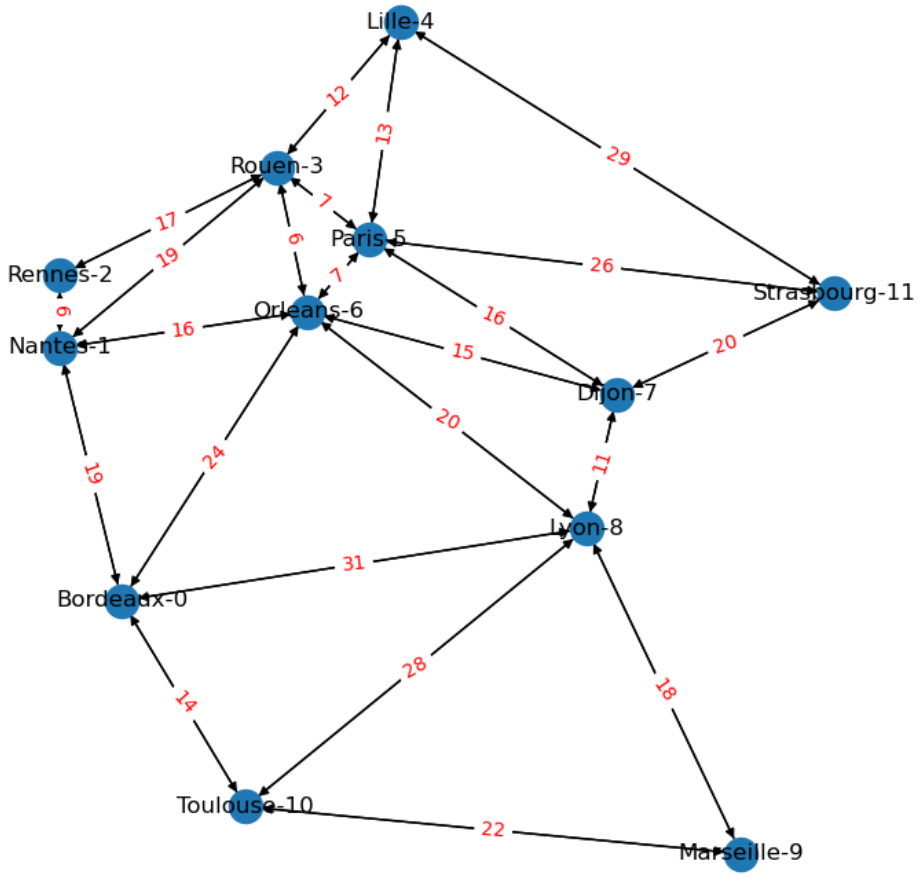
TP N°07 : GRAPHES PONDERES ET ALGORITHMES DE PLUS COURT CHEMIN

OBJECTIFS DU TP

- Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs avec l'algorithme de Dijkstra
- Mettre en évidence qu'il est possible d'améliorer sa rapidité avec une heuristique.

Document : Bordeaux – Strasbourg en vélo ?

Toujours à la recherche de nouveaux exploits à accomplir, vous décidez de partir de Bordeaux pour rejoindre Strasbourg en empruntant les chemins décrits sur cette carte. Une connexion est faite entre les chefs-lieux des régions contiguës. Le chiffre associé représente le **nombre d'heures à vélo** (sans les pauses...).

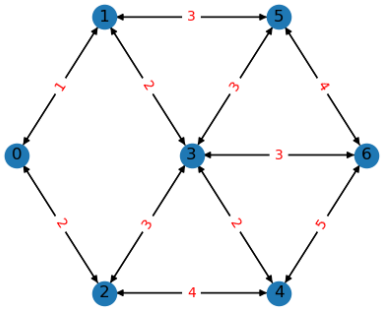


Q1. Trouver « à la main » sans stratégie particulière, le plus court chemin entre Bordeaux et Strasbourg.

Le chemin le plus court est : Bordeaux – Orleans – Paris – Strasbourg.

I. Recherche du plus court chemin avec l'algorithme de Dijkstra

Pour (re)prendre en main l'algorithme de Dijkstra, on va dans un premier temps se concentrer sur un graphe d'ordre inférieur, ce qui va nous permettre d'appliquer l'algorithme « à la main » dans un temps raisonnable.



Q2. Sans l'implémenter, trouver le plus court chemin en utilisant manuellement l'algorithme de Dijkstra entre les sommets 0 et 6 sur le graphe ci-dessus. Noter chaque étape de raisonnement dans le tableau ci-dessous pour vous familiariser avec l'algorithme. L'algorithme est rappelé ci-dessous.

Étape	Listes E des sommets à parcourir	S	Voisins de S	Distance à D							Prédécesseurs						
				0	1	2	3	4	5	6	0	1	2	3	4	5	6
Ini	0,1,2,3,4,5,6			∞	∞	∞	∞	∞	∞	∞	N	N	N	N	N	N	N
1	1,2,3,4,5,6	0	1,2	**0**	1	2	∞	∞	∞	∞	N	0	0	N	N	N	N
2	2,3,4,5,6	1	0,3,5	-	**1**	2	3	∞	4	∞	N	**0**	0	1	N	1	N
3	3,4,5,6	2	0,3,4	-	-	**2**	5	6	4	∞	N	-	**0**	2	2	1	N
4	4,5,6	3	1,2,4,5,6	-	-	-	**3**	5	6	6	N	-	-	**1**	3	3	3
5	4,6	5	1,3,6	-	-	-	-	5	**4**	8	N	-	-	-	3	**1**	5
6	6	4	2,3,6	-	-	-	-	**5**	-	11	N	-	-	-	**3**	-	4
7		6	3,4,5	-	-	-	-	-	-	**6**	N	-	-	-	-	-	**3**
				0	1	2	3	5	4	6	N	0	0	1	3	1	3

La plus courte durée entre les sommets 0 et 6 est 6. Pour celui il faut suivre le chemine suivant : 0-1-3-6.

Q3. Ecrire une fonction **`voisins`** qui prend en argument une matrice d'adjacence M d'un graphe pondéré et un entier S et renvoie la liste sommets adjacents au sommet S.

```
def voisins(G,i):
    """
    détermine de nombre de sommets adjacents à un sommet si dans un graphe de matrice d'adjacence G
    entrée : une matrice d'adjacence d'un graphe G et un entier i correspondant à un sommet du graphe
    sortie : liste L contenant les numéros des sommets adjacents au sommet si

    ===
    jeux de tests
    ===
    G=[[0,1,1],[1,0,0],[1,0,0]]
    voisins(G,0)
    >>> [1,2]
```

```
    voisins(G,2)
    >>> [0]
    """
    n = len(G)
    L = []
    # Lecture de la ligne i de la matrice d'adjacence de G
    for j in range(n):
        if G[i][j] < inf:
            L = L + [j]
    return L
```

Q4. Ecrire une fonction **Dijkstra** prenant en argument une matrice M d'adjacence du graphe pondéré G(S,A) et deux entiers D et A représentant respectivement les sommets de départ et d'arrivée et retournant la liste des sommets correspondant au plus court chemin pour aller du départ à l'arrivée et un entier correspondant à la distance de ce chemin. On pourra utiliser la fonction **voisins**.

```
def Dijkstra (M:[list],D,A)->list:
    """
    revoie le plus court chemin ainsi que la distance parourue pour aller du sommet n°D au sommet n°A sur le graphe pondéré de matrice d'adjacence M en utilisant l'algorithme de Dijkstra

    entrée : M matrice d'adjacence d'un graphe pondéré représentant la carte
    A et D : entiers correspondant au sommet de départ et à celui d'arrivée

    sortie : liste d'entiers représentant les sommets visités sur le plus court chemin entre D et A
    un entier représentant la distance parcourue entre D et A par le plus court chemin

    """

    n = len(M)
    E = [sommet for sommet in range(n)] # sommets qui restent à explorer

    # liste telle que dist [i] correspond à la distance du plus court chemin entre le sommet D et le sommet i, initialisée à l'infini.
    dist = [inf for _ in range (n)]

    # liste telle que pred[i] est un entier correspondant au prédécesseur du sommet i par le plus court chemin entre D et i
    pred = [None for _ in range (n)]

    S = D # on commence par le point de départ
    E.remove(D) # on enlève le départ des sommets à explorer
    dist[D]=0

    # ajout pour le suivi des sommets visités (ajout pour Q6)
    visit = [D]

    while E != []:
```

```
        V = voisins(M,S)
        for v in V:
            # on cherche la plus petite distance entre D et v en passant par S
            cout = dist[S] + M[S][v]
            if v in E and cout < dist[v]:
                dist[v]=cout
                pred[v]=S

            # on repart du sommet S pour lequel il y a le plus court chemin entre
S et D
        mini=inf
        for s in range (n):
            if s in E and dist[s]<mini :
                S = s
                mini = dist[s]
        visit.append(S) # ajout pour Q6
        E.remove(S)


    # on utilise la liste des prédécesseurs pour trouver le chemin correspondant
à la plus courte distance
        chemin = [A]
        P = pred[A]
        while P != None:
            chemin = [P]+chemin
            P=pred[P]

    return(dist[A],chemin,visit)
```

Q5. Utiliser la fonction `Dijkstra` pour trouver le plus court chemin ainsi que sa durée entre Bordeaux et Strasbourg.

```
distance,chemin =Dijkstra(Carte_M,0,11)
print("la distance minimale est",distance)
print("le chemin le plus court est",chemin)

>>> la distance minimale est 57
le chemin le plus court est [0, 6, 5, 11]
```

Q6. Modifier la fonction précédente pour qu'elle renvoie également la liste de tous les sommets explorés pour trouver le plus court chemin.

Il suffit d'ajouter un compteur (voir les ajouts en gris à la Q4).

Q7. Ecrire une instruction permettant de créer une liste `H` contenant à l'indice i l'heuristique proposée pour le sommet i.

```
H = [47, 44, 43, 31, 25, 25, 27, 15, 24, 39, 46, 0]
```

Q8. Ecrire une fonction `A_star` prenant en argument une matrice M d'adjacence du graphe pondéré G(S,A), deux entiers D et A représentant respectivement les sommets de départ et

d'arrivée et une liste H représentant l'heuristique retenue. Cette fonction retourne la liste des sommets correspondant à un des plus courts chemins pour aller du départ à l'arrivée.

Remarque : l'algorithme s'arrête dès qu'une solution est trouvée. Ce n'est pas nécessairement la meilleure mais souvent l'une des meilleures.

Très peu de modifications ont été effectuées par rapport à l'algorithme précédent.

```
def A_star (M:[list],D,A,H)->list:
    """
    revoie un des plus court chemin ainsi que la distance parourue pour aller du sommet n°D au sommet n°A sur le graphe pondéré de matrice d'adjacence M en utilisant l'algorithme de Dijkstra

    entrée : M matrice d'adjacence d'un graphe pondéré représentant la carte
    A et D : entiers correspondant au sommet de départ et à celui d'arrivée
    H : une liste contenant une heuristique

    sortie : liste d'entiers représentant les sommets visités sur le plus court chemin entre D et A
    un entier représentant la distance parcourue entre D et A par le plus court chemin
    """

    n = len(M)
    E = [sommet for sommet in range(n)] # sommets qui restent à explorer

    # liste telle que distance[i] correspond à la distance du plus court chemin entre le sommet D et le sommet i, initialisée à l'infini.
    dist = [inf for _ in range (n)]

    # liste telle que predesceur[i] est un entier correspondant au prédéscesseur du sommet i par le plus court chemin entre D et i
    pred = [None for _ in range (n)]

    S = D # on commence par le point de départ
    E.remove(D) # on enlève le départ des sommets à explorer
    dist[D]=0

    # ajout pour le suivi des sommets visités
    visit = [D]

    while dist[A]==inf and E!=[]: # on s'arrête dès qu'on a trouvé un chemin
        V = voisins(M,S)
        for v in V:
            # on cherche la plus petite distance entre D et le sommet v en passant par S
            cout = dist[S] + M[S][v]
            if v in E and cout < dist[v]:
                dist[v]=cout
                pred[v]=S
```

```
            # on repart du sommet S qui parmi les voisins a le plus de chance de
se trouver sur le plus court chemin, en utilisant l'heuristique H

        mini=inf
        for s in range (n):
            dist_corr=dist[s]+H[s] # somme de la plus petite durée parcourue
jusqu'à ce sommet et de la durée à vol d'oiseau de la destination
            if s in E and dist_corr<mini :
                S = s
                mini = dist_corr

        visit.append(S) # ajout pour Q9
        E.remove(S)

    # on utilise la liste des prédécesseurs pour trouver le chemin correspondant
à la plus courte distance

        chemin = [A]
        P=pred[A]
        while P != None:
            chemin = [P]+chemin
            P=pred[P]

    return(dist[A],chemin,visit)
```

Q9. Modifier la fonction précédente pour qu'elle retourne également la liste de tous les sommets visités.

Voir ajouts en gris clair.

Q10. Commenter en comparant les deux algorithmes.

Avec les deux algorithmes, on trouve ici le même « plus court chemin ». L'algorithme A* est cependant plus rapide et permet de visiter moins de sommets. On voit ici l'intérêt de l'heuristique choisie.

Avec le script suivant, il est possible de mettre en rouge les sommets visités par l'algorithme pour trouver le plus court chemin :

```
# graphe de base (pour avoir les couleurs ET les noms)
draw(G,with_labels=True,pos=pos)
labels = get_edge_attributes(G,'weight')
draw_networkx_edge_labels(G,pos,edge_labels=labels,font_color='r',font_size=10)
couleur=array('r' for _ in range (len(G)))
plt.show()

#coloration des sommets visités

sommets_parcourus = [0]*len(Carte_M) #sommet_parcourus[i]=0 si le sommet n'a pas
été visité et 1 sinon

for v in visite :
```

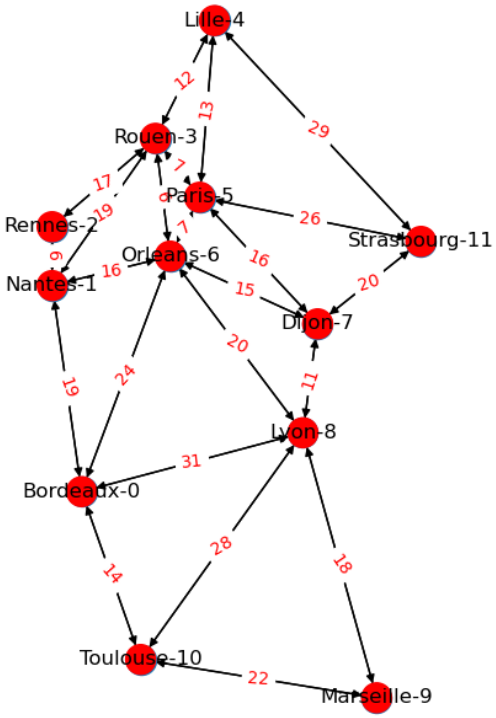
```
    sommets_parcourus[v]=1

# position des sommets (pour qu'ils soient superposés aux autres)
pos_M={}
i=0
for position in pos.values() :
    pos_M[i]=list(position)
    i+=1
print(pos_M)

#ajout des couleurs
couleur=[]
for sommet in sommets_parcourus :
    if sommet==1:
        couleur.append("r")
    else :
        couleur.append("b")
color=array(couleur)

Carte = from_numpy_array(array(Carte_M))
draw_networkx_nodes(Carte, node_color = color,pos=pos_M)
plt.axis("equal")
plt.show()
```

Sommets visités avec l'algorithme de Dijkstra



Sommets visités avec l'algorithme A* en utilisant une heuristique