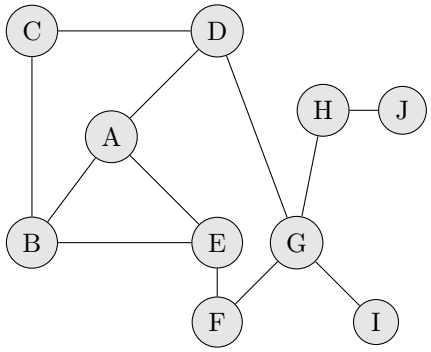
TP n° 14 – Graphes

I Parcours de graphes

Dans ce TP, les graphes seront implémentés sous forme de dictionnaires. Par exemple, on considère le graphe suivant :



Graphe `gr`

Le dictionnaire qui le représente commence comme :

```
gr = {'A' : ['B' , 'D' , 'E'], 'B' : ['A' , 'C' , 'E'], ...}
```

Exercice 1. Écrire le dictionnaire complet associé au graphe `gr`.

Solution 1. `gr={'A':['B','D','E'],'B':['A','C','E'],'C':['B','D'],'D':['A','C','G'], 'E':['A','B','F'],'F':['E','G'],'G':['D','F','H','I'],'H':['G','J'],'I':['G'],'J':['H']}`

Exercice 2. Parcours en largeur.

1. Écrire une fonction `ParcoursLargeur` qui étant donné un graphe et un sommet de ce graphe, réalise un parcours en largeur et retourne la liste des sommets dans l'ordre dans lequel ils ont été explorés.
2. Tester votre fonction sur le graphe donné en exemple.
   `ParcoursLargeur(gr,'A')` renvoie `['A', 'B', 'D', 'E', 'C', 'G', 'F', 'H', 'I', 'J']`.
3. Modifier la fonction `ParcoursLargeur` afin qu'elle affiche les nœuds à explorer à chaque étape.
4. Commenter si cette suite d'élément se comporte en `pile` ou en `file`.

Solution 2.

```
def PL2(gr,s):
    files=[s]
    noeuds_decouverts=[s]
    while len(files)!=0:
        print(files)
        noeud_courant=files[0]
        files=files[1:]
        for voisin in gr[noeud_courant]:
            if voisin not in noeuds_decouverts:
                noeuds_decouverts.append(voisin)
                files.append(voisin)
    return(noeuds_decouverts)
```

Les noeuds entrent par la fin et sortent par l'avant, il s'agit donc d'une file.

Exercice 3. Parcours en profondeur.

1. Écrire une fonction `ParcoursProfondeur` qui étant donné un graphe et un sommet de ce graphe, réalise un parcours en profondeur et retourne la liste des sommets dans l'ordre dans lequel ils ont été explorés.

2. Tester votre fonction sur le graphe donné en exemple.
   `ParcoursProfondeur(gr,'A')` renvoie `['A', 'E', 'F', 'G', 'I', 'H', 'J', 'D', 'C', 'B']`.
3. Modifier la fonction `ParcoursProfondeur` afin qu'elle affiche les nœuds à explorer à chaque étape.
4. Commenter si cette suite d'élément se comporte en `pile` ou en `file`.

Solution 3.

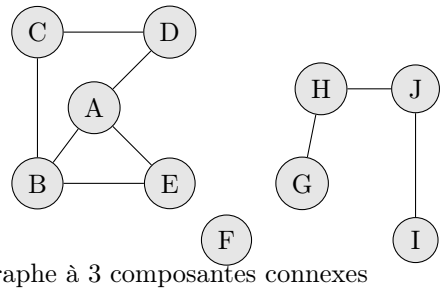
```
def PP2(gr,s):
    piles=[s]
    noeuds_decouverts=[]
    while len(piles)!=0:
        print(piles)
        noeud_courant=piles[-1]
        piles=piles[:-1]
        for voisin in gr[noeud_courant]:
            if (voisin not in noeuds_decouverts) and (voisin not in piles):
                piles.append(voisin)
        noeuds_decouverts.append(noeud_courant)

    return(noeuds_decouverts)
```

Les noeuds entrent par la fin et sortent par la fin, il s'agit donc d'une pile.

Exercice 4. Composantes connexes.

1. Écrire une fonction `EstConnexe` qui prend un graphe comme argument et renvoie `True` si le graphe est connexe et `False` sinon. Par exemple, pour le graphe précédent, `EstConnexe(gr)` renvoie `True`.
2. Écrire une fonction `NbCompConnexe` qui prend un graphe comme argument et renvoie le nombre de composantes connexes de ce graphe.
   Par exemple, pour le graphe suivant, la fonction doit renvoyer 3.



Graphe à 3 composantes connexes

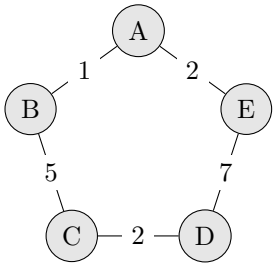
Solution 4.

```
def EstConnexe(gr):
    sommet=gr.keys()[0]
    liste=PL(gr,sommet)
    return(len(liste)==len(gr.keys()))


def NbCompConnexe(gr):
    liste_sommets=gr.keys()
    Nb=0
    while len(liste_sommets)!=0:
        Nb=Nb+1
        sommet=liste_sommets[0]
        for voisin in PL(gr,sommet):
            liste_sommets.remove(voisin)
    return(Nb)
```

II Algorithme de Dijkstra

On considère un graphe connexe pondéré avec des poids positifs. L'objectif est de trouver le chemin entre deux sommets qui minimise la somme des poids rencontrés.



Un tel graphe peut être représenté par un dictionnaire de dictionnaires. Celui ci-dessus commence par :

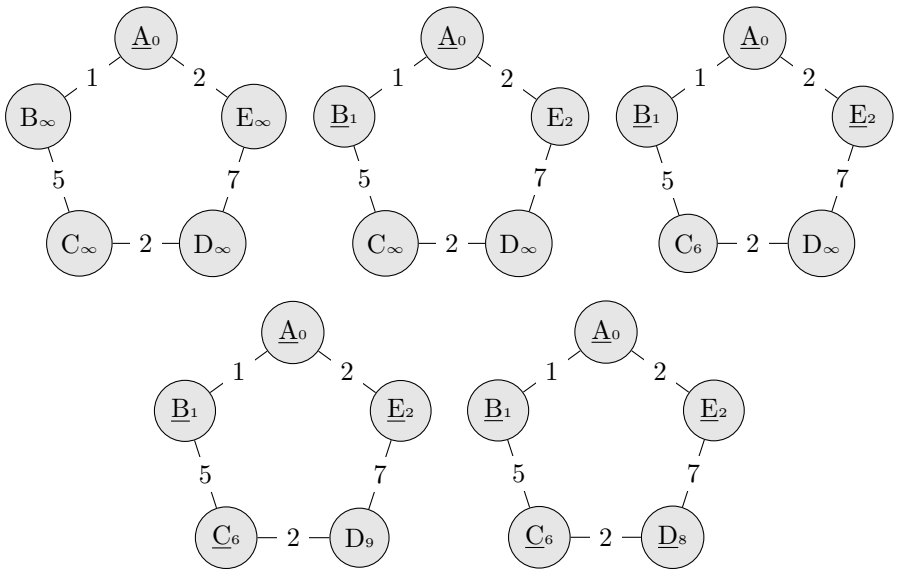
```
gr = {'A' : { 'B' : 1 , 'E' : 2 } , ...}
```

L'algorithme de Dijkstra permet de répondre à ce problème selon le principe suivant :
au départ, la distance au sommet A est nulle et toutes les autres distances sont infinies. Le sommet A est alors exploré.
A chaque itération :

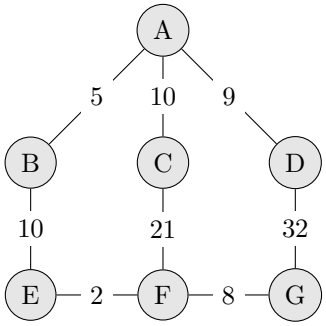
- on découvre les sommets adjacents ;
- la distance d'un trajet est égale à la somme des poids des arcs ;
- on choisit le prochain sommet à explorer comme celui qui est à distance minimale du sommet de départ.

On s'arrête quand tous les sommets ont été explorés.

On applique l'algorithme sur l'exemple précédent pour minimiser la distance entre A et D (on souligne à chaque étape la distance minimale pour choisir le nœud à explorer) :



Exercice 5. Appliquer à la main l'algorithme de Dijkstra au graphe ci-dessous entre les sommets A et G.



Solution 5.

noeuds à explorer	à A	à B	à C	à D	à E	à F	à G
A	$\underline{0}$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
B	-	$\underline{5}$	10	9	$+\infty$	$+\infty$	$+\infty$
D	-	-	10	$\underline{9}$	15	$+\infty$	$+\infty$
C	-	-	$\underline{10}$	-	15	$+\infty$	41
E	-	-	-	-	$\underline{15}$	31	41
F	-	-	-	-	-	$\underline{17}$	41
G	-	-	-	-	-	-	25

Chemin minimal : `'A'-'B'-'E'-'F'-'G'`

Exercice 6. Dans l'algorithme, à chaque itération, on a besoin de choisir le chemin le plus court parmi ceux déjà explorés. Écrire une fonction `MinDistance` qui à un dictionnaire du type `{'A' : 1, 'B' : 4, 'C' : 0}` renvoie le sommet de poids minimal.

Solution 6.

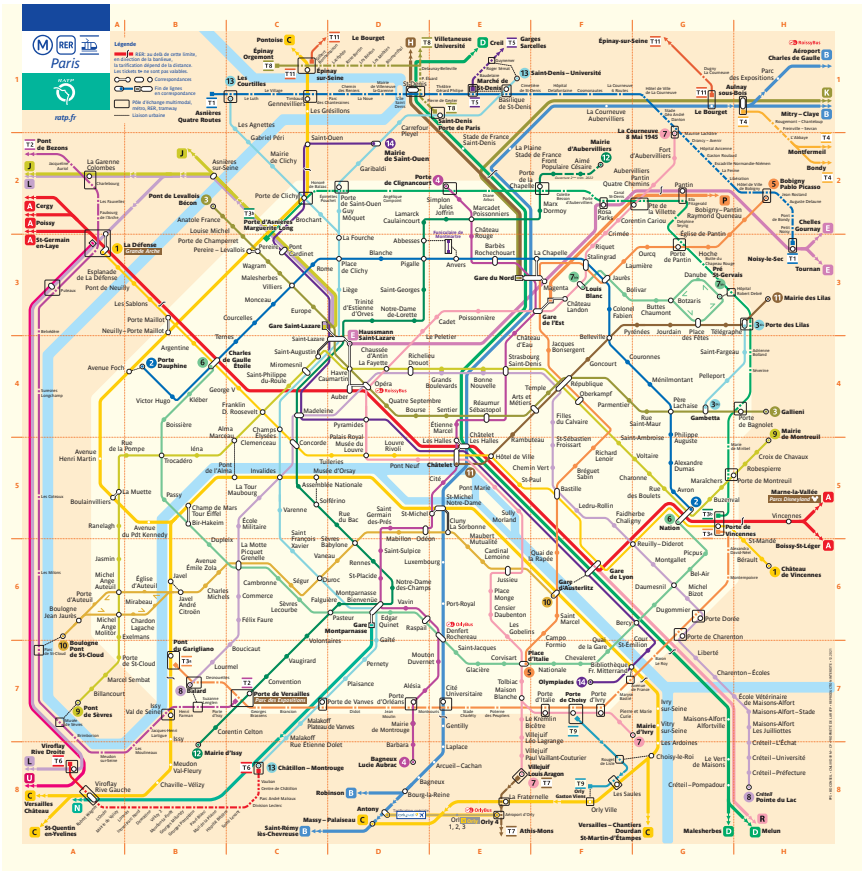
```
def MinDistance(sousgr):
    sommet_mini=sousgr.keys()[0]
    mini=sousgr[sommet_mini]
    for sommet in sousgr.keys():
        if sousgr[sommet]<mini:
            mini=sousgr[sommet]
            sommet_mini=sommet
    return(sommet_mini)
```

Exercice 7. Ouvrir le fichier CodeDijkstra.py dans le lequel se trouve la fonction `Dijkstra` (version récursive), rédigée par Frank Chevrier. Elle prend comme entrée un graphe pondéré (présenté sous forme de dictionnaire) et deux sommets du graphe et renvoie le chemin le plus court. Tester cette fonction sur un graphe pondéré.

Solution 7.

III Application : temps de trajet en métro

D'après une idée de Michel Couprie et Gilles Bertrand



On considère le plan du métro parisien. Chaque sommet correspond à une station pour une ligne. Par exemple, il y a 5 sommets "République". Deux sommets sont reliés par une arête pondérée par le temps de trajet en seconde entre les deux stations. Il existe aussi une arête entre les stations de même nom. Ces arêtes sont toutes pondérées par 120 (autrement dit, on compte 120s pour effecteur une correspondance).

Exercice 8.
1. Combien d'arêtes partent du sommet "Croix de Chavaux" ? d'un des sommets "République" ?
2. Expliquer pourquoi le graphe est orienté.

Solution 8.
1. Deux arêtes partent de "Croix de Chavaux". L'une la relie à 'Mairie de Montreuil', l'autre à 'Robespierre'.
   Depuis l'un des sommet "République", on a deux arêtes qui partent pour aller aux stations voisines sur la même ligne et on a 4 arêtes qui relient aux autres sommets 'République'.
2. La ligne 7 bis par exemple ne pas pas être parcourue de façon symétrique.

Dans le fichier data_nom.csv, on trouve les noms des stations, avec chacune un numéro. Dans le fichier data_arete.csv, chaque ligne contient trois nombres : le numéro de la station de départ, le numéro de la station d'arrivée, le temps de trajet.

Exercice 9.
1. Importer les deux fichiers sous forme de deux tableaux : le premier tableau aura deux colonnes (entier, chaîne de caractères), le deuxième trois colonnes (entier, entier, flottant).
2. Construire le dictionnaire associé à ce graphe pondéré.
3. Appliquer la fonction `Dijkstra` à ce graphe, entre deux stations.

Solution 9.

```
fichier = open("data/data_nom.csv", "r")
fichier2 = open("data/data_arete.csv", "r")

tableau=fichier.read()
```

```
tableau2=fichier2.read()

fichier.close()
fichier2.close()

lignes=tableau.split('\n')
nom=[]
for i in lignes[:-1]:
    ligne=i.split(',')
    ligne[0]=int(ligne[0])
    nom.append(ligne)

lignes2=tableau2.split('\n')
arete=[]
for i in lignes2[:-1]:
    ligne=i.split(',' )
    arete.append([int(ligne[0]),int(ligne[1]),float(ligne[2])])

dico={}
for i in arete:
    if i[0] not in dico:
        dico[i[0]]={i[1]:i[2]}
    else:
        dico[i[0]][i[1]]=i[2]

Trajet=Dijkstra(dico,88,302)

print(Trajet[0]/60.)
for i in Trajet[1]:
    print(nom[i])
```