```
"""
Authors : Nicolas DARGAZANLI et Alexandre MAURICE, en TD 1 TP 1
Remarque : A* n'est pas fonctionnel mais les traces du code sont laissées
"""

import pandas as pd
from math import sin, cos, acos, pi
import numpy as np
import time as t
donneesbus=pd.read_csv(r'./donneesbus.csv',sep=';')

arrets={}
for c in range (len( donneesbus)):
    arrets[donneesbus['arret'][c]]=[float(donneesbus['lattitude']
[c].replace(",",".")),float(donneesbus['longitude'][c].replace(",",".")),list(donneesbus['listesucc']
[c].replace('[',").replace(']',").replace("'",").replace(" ",").split(","))]

nom=list(arrets.keys())

nom_arrets=[]
for i in arrets:
    nom_arrets.append(i)

def nom(ind):
    return nom_arrets[ind]

def indice_som(nom_som):
    return nom_arrets.index(nom_som)

def latitude(nom_som):
    return arrets[nom_som][0]

def longitude(nom_som):
    return arrets[nom_som][1]

def coordonnes(nom_som):
    return latitude(nom_som),longitude(nom_som)

def voisin(nom_som):
    return arrets[nom_som][2]

def dic_adjacence(donnees):
    dic={}                                                  #Création d'un dictionnaire vide
    for i in donnees:                                       #Récupération des clés du dictionnaires
"donnees"
        dic[i]=donnees[i][2]                                    #Récupération des arrêts succedant
de l'arrêts i
    return dic                                              #Retour du dictionnaire d'adjacence crée


def lst_adjacence(donnees):
    """Cette fonction renvoie une matrice d'ajacence à partir d'un dictionnaire d'adjacence
```

```
    """
    lst = [[0]*len(donnees) for _ in range(len(donnees))]              #Création d'un tableau à double entrées initialisé à 0 pour tous les arrêts

    for c in arrets:                                                   #Pour tous les arrêts dans le dictionnaires des arrêts
        succ=voisin(c)                                                 #On enregistre la liste des arrêts succedant de l'arrêt c
        for i in succ:                                                 #Pour tous ses successeurs
            lst[indice_som(c)][indice_som(i)]=1                        #On ajoute l'adjacence entre les deux arrêts
    return lst                                                         #Retour de la matrice d'adjacence

def distanceGPS(latA,latB,longA,longB):
    # Conversions des latitudes en radians
    ltA=latA/180*pi
    ltB=latB/180*pi
    loA=longA/180*pi
    loB=longB/180*pi
    # Rayon de la terre en mètres (sphère IAG-GRS80)
    RT = 6378137
    # angle en radians entre les 2 points
    S = acos(round(sin(ltA)*sin(ltB) + cos(ltA)*cos(ltB)*cos(abs(loB-loA)),14))
    # distance entre les 2 points, comptée sur un arc de grand cercle
    return S*RT

def distarrets(arret1,arret2):
    """Cette fonction retourne la distance en mètres entre deux arrets grâce a leurs coordonnées GPS"""
    lat1=latitude(arret1)       #Recuperation de la latitude de l'arret1
    lat2=latitude(arret2)       #Recuperation de la latitude de l'arret2
    long1=longitude(arret1)     #Recuperation de la longitude de l'arret1
    long2=longitude(arret2)     #Recuperation de la longitude de l'arret2
    return distanceGPS(lat1,lat2,long1,long2) #Appel et retour du résultat de la fonction de calcule à vol d'oiseau de deux points GPS


def distarc(arret1,arret2):
    """Cette fonction renvoie la distance en mètres entre deux arrêts donnés en paramètres:
            - Si l'arret2 est un successeur de l'arret1 le retour sera la distance a vol d'oiseau entre ces deux arrêts
            - Si l'arret2 n'est pas un successeur de l'arret1 le retour sera une distance infinie
    """
    if arret2 in voisin(arret1) or arret1==arret2 :             #Si l'arret2 est un successeur de l'arret1
        res=distarrets(arret1,arret2)          #Appel de la fonction calculant la distance des deux arrêts
    else:                                  #Sinon
        res=np.Inf                         #La distance est dite Infinie
    return res

mat_bus=lst_adjacence(arrets)
poids_bus=[x[:] for x in mat_bus]                  #Copie profonde de la matrice d'adjacence qui possède le même ordre
```

```
for i in range (len(poids_bus)):                    #On parcours la matrice poids_bus
    for y in range(len(poids_bus[i])):
        poids_bus[i][y]=distarc(nom(i),nom(y))        #On y inscrit la distance entre les deux arrêts
selectionnés [floatant ou Infini]


#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~DEBUT
DJIKSTRA~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~



def min_exclude(distance,marque):
    """Retourne l'indice du tableau de la valeur minimale en ignorant les valeurs des indices inscrit
dans la liste marque"""
    min=-1                             #valeur par default marqueur d'erreur si aucun arret est trouvé
    for i in range (len(distance)):     #On parcours tout les arrets
        if i not in marque:                    #S'il n'est pas déjà marqué
            if min==-1:
                min=i                             #La première valeur devient le minimum
            elif distance[i][0]<distance[min][0]:   #Si l'arret i à une distance inferieur a l'arret min
                min=i                             #L'arret i devient le nouveau min
    return min         #Retourne l'arret avec la distance minimale

def dijkstra(depart,arrive):
    '''Cette fonction prend en paramètres deux arrêts et renvoie le plus court chemin, sous forme de la
liste des arrêts parcourus ainsi, que la distance minimum en
    utilisant la méthode de Djiksrta.'''
    # Initialisation
    distance=[(np.Inf,None) for _ in range(len(nom_arrets))]
    distance[indice_som(depart)]=(0,indice_som(depart))               # On ajoute la distance de l'arret
de départ soit 0, son pred est lui même
    marque=[indice_som(depart)]                                  # Liste de tous les arrets dont la distance
minimum a déjà été trouvée
    arret_actuel=indice_som(depart)                              # arret_actuel est l'arret a partir du quel
on va effectuer l'étape

    while arret_actuel!=indice_som(arrive):                        # On peut raccourcir djikstra en
s'arretant dès que l'on doit traiter le sommet d'arrivée
        for proche in voisin(nom(arret_actuel)):                                                   #Pour
tous les arrets voisins de l'arret observé
            if indice_som(proche) not in marque:                                                   #Si
l'arret n'est pas déjà marqué (on pourra pas l'ameliorer de toute facon)
                if distance[indice_som(proche)][0]>distance[arret_actuel]
[0]+distarc(nom(arret_actuel),proche):              #Si ce nouveau chemin est plus avantageux que
l'ancien
```

```
            distance[indice_som(proche)]=(distance[arret_actuel]
[0]+distarc(nom(arret_actuel),proche),arret_actuel)   #On met a jour sa distance et son prédécesseu
        arret_actuel=min_exclude(distance,marque)                  #On récupere l'arret avec la distance
minimale parmis les arrets non marqués
        marque.append(arret_actuel)                                #On ajoute l'arret actuel dans les arrets
marqués


    # Reconstruction
    arret_actuel=indice_som(arrive)                             #On parcours les arrets en commançant par
l'arrivée
    chemin=[nom(arret_actuel)]                                   #On crée une liste de allant de l'arrivée vers
le depart
    while arret_actuel!=indice_som(depart):                    #Tant que l'arret actuel n'est pas le départ
on continue le chemin
        pred=distance[arret_actuel][1]                       #Prédécesseur de l'arret actuel
        arret_actuel=pred                                 #L'arret actuel devient le Prédécesseur
        chemin.append(nom(arret_actuel))                     #On ajoute le Prédécesseur au chemin

    chemin.reverse()                                   #On inverse la liste pour obtenir le chemin dans le
bon ordre (départ vers arrivée)
    return (chemin,round(distance[indice_som(arrive)][0]))      # Renvoie : chemin (liste), distance
(entier)


#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~FIN
DJIKSTRA~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~DEBUT
FORD~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~

def difference(nouvelle,ancienne):
    """Renvoie les indices des valeurs differentes dans une liste
        - len(liste1)==len(liste2)"""
    diff=[]
    for y in range(len(nouvelle)):
        if nouvelle[y]!=ancienne[y]: #Si les deux valeurs (ici des distances) sont différentes
            diff.append(y)          #On ajoute l'indice i (representant un arret) à la liste de retours
```

```
    return diff                    #On retourne la liste de tout les indices(arrets) ayant une valeur(distance) 
changée


def ford(depart,arrive):
    """Cette fonction prend en paramètres deux arrêts et renvoie le plus court chemin, sous forme de 
la liste des arrêts parcourus ainsi, que la distance minimum en utilisant la
    méthode de Bellman Ford-Kalaba."""
    distance=[(np.Inf,None) for _ in range(len(nom_arrets))]    #Tous les arrets ont une distances 
infinie et pas de predécesseur
    distance[indice_som(depart)]=(0,indice_som(depart))        #On ajoute la distance de l'arret de 
départ soit 0, son pred est lui même
    a_traiter=[indice_som(depart)]                          #Une liste de tous les arrets que l'ont doit traiter
    while a_traiter!=[]:                              #Tant que cette liste n'est pas vide on continue la 
recherche, on assume donc qu'il n'y a pas de chemins absorbants
        distancepred=distance[:]                          #Copie de la liste de distance afin de la comparer 
après l'étape

        for arret_actuel in a_traiter:                                                 #L'arrets en cours de 
traitement
            for proche in voisin(nom(arret_actuel)):                                            #Pour tous les 
arrets voisins de l'arret observé
                if distance[indice_som(proche)][0]>distance[arret_actuel]
[0]+distarc(nom(arret_actuel),proche):        #Si ce nouveau chemin est plus avantageux que l'ancien
                    distance[indice_som(proche)]=(distance[arret_actuel]
[0]+distarc(nom(arret_actuel),proche),arret_actuel)     #On met a jour sa distance et son 
prédécesseur
        a_traiter=difference(distance,distancepred)           #on récupere tous les arrets qui ont était mis 
a jour dans l'étape
                                                  #Opération faite après l'étape pour ne pas interferer avec les 
données de l'étape

    # Reconstruction
    chemin=[arrive]                              #On crée une liste de allant de l'arrivée vers le depart
    arret_actuel=indice_som(arrive)                        #On parcours les arrets en commançant par 
l'arrivée
    while arret_actuel!=indice_som(depart):                  #Tant que l'arret actuel n'est pas le départ 
on continue le chemin
        pred=distance[arret_actuel][1]                     #Prédécesseur de l'arret actuel
        chemin.append(nom(pred))                          #On ajoute le Prédécesseur au chemin
        arret_actuel=pred                              #L'arret actuel devient le prédécesseur
    chemin.reverse()                              #On inverse la liste pour obtenir le chemin depart-
>arrivée
    return (chemin,round(distance[indice_som(arrive)][0]))          #On retourne le chemin et la 
distance entre ces deux arrets

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~FIN 
FORD~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

def floyd(depart,arrive):
    """Cette fonction prend en paramètres deux arrêts et renvoient le plus court chemin, sous forme de la liste des arrêts parcourus ainsi, que la distance minimum en utilisant la
    méthode de floyd wharshall."""
    M0=[x[:] for x in poids_bus]                    #Initialisation de M0
    P0=[x[:] for x in poids_bus]                    #Initialisation de P0
    #Initialisation des prédécesseurs
    for i in range (len(P0)):
        for j in range(len(P0)):
            if P0[i][j]==np.Inf or P0[i][j]==0:
                # Si +infini ou diagonale : Pas de prédécesseur connu ou existant
                P0[i][j]=None
            else:
                # Sinon, le prédécesseur est l'indice de la colonne
                P0[i][j]=i
    #------------------------------------

    n=len(nom_arrets) # Taille de la matrice
    k=0 # Numéro de l'étape (= tous les chemins de longueur <= k)
    while k < n : # Condition d'arrêt : dépassement de la taille de la matrice
        MN=[x[:] for x in M0] # Copie profonde de M0 vers MN
        # Note importante : par la suite M0 représente MN et MN représente MN+1, n entier naturel
        PN=[x[:] for x in P0] # Copie profonde de PO vers PN
        # Même remarque
        for i in range (n):
            for j in range (n):
                # Application de la formule permettant de passer de PN à PN+1
                if i!=j and M0[k][j]+M0[i][k]<M0[i][j]:
                    MN[i][j]=M0[k][j]+M0[i][k]
                    PN[i][j]=P0[k][j]
                else:
                    MN[i][j]=M0[i][j]
                    PN[i][j]=P0[i][j]
        # Copies profondes
        M0=[x[:] for x in MN]
        P0=[x[:] for x in PN]
        # Remarque : On écrase les matrices car toutes les informations sont contenues dans les matrices finales
        k+=1
    # Reconstruction
    arret_actuel=indice_som(arrive)
    chemin=[nom(arret_actuel)]
    while arret_actuel!=indice_som(depart):

        pred=P0[indice_som(depart)][arret_actuel]
        chemin.append(nom(pred))
        arret_actuel=pred
    chemin.reverse()
```

```
    return (chemin,round(M0[indice_som(depart)][indice_som(arrive)]))

"""
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Tentative pour A*...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

def filePrioritaire(file):
    min=np.Inf
    for i in range(len(file)):
        if file[i][1]<=min:
            min=file[i][1]
    stock=file[i]
    file.pop(i)
    return file[i]

def compareParHeuristique(n1,n2):
    if n1[1]<n2[1]:
        return 1
    elif n1[1]==n2[1]:
        return 0
    else :
        return -1



def f(n,actualCost,arrivee):
    return distarrets(n,arrivee)+actualCost

def astar(depart,arrivee):
    openList = [depart]
    closedList=[]
    heuristique={}
    heuristique[depart]=0
    parcours={}
    parcours[depart]=depart
    while len(openList) > 0:
        n = None
        for arret in openList:
            if n == None or parcours[arret]+f(arret,heuristique[arret],arrivee) < heuristique[n]
+f(n,heuristique[n],arrivee):
                n=arret
        if n == arrivee:
            reconstruction=[]
            while parcours[n] != n :
                reconstruction.append(n)
                n=parcours[n]
            reconstruction.append(depart)
            reconstruction.reverse()
            return reconstruction
        for voisinActuel in voisin(n):
            print(parcours[n])
            poids=parcours[n]
```

```
            print(heuristique)
            if voisinActuel not in openList and voisinActuel not in closedList:
                openList.append(voisinActuel)
                parcours[voisinActuel] = n
                heuristique[voisinActuel] = heuristique[n] + poids
        openList.remove(n)
        closedList.append(n)
        return None
"""

def test():
    """Programme permettant de tester le temps d'exécution des différentes fonctions
       On vérifie aussi si le résultat renvoyé le même
       Remarque : l'exécution du test peut dépasser la minute"""
    tmp_dijkstra = t.time()
    res_dijkstra = dijkstra("NOVE","TROICR")
    duree_dijkstra = t.time() - tmp_dijkstra
    print("Dijkstra :",duree_dijkstra)

    tmp_ford = t.time()
    res_ford = ford("NOVE","TROICR")
    duree_ford = t.time() - tmp_ford
    print("Bellman-Ford Kalaba :",duree_ford)

    tmp_floyd = t.time()
    res_floyd = floyd("NOVE","TROICR")
    duree_floyd = t.time() - tmp_floyd
    print("Floyd-Warshall :",duree_floyd)

    if res_dijkstra==res_ford and res_dijkstra==res_floyd:
        print("Les trois algorithmes renvoient bien le même résultat.")

test()
```