MINI PROJET : DIJKSTRA

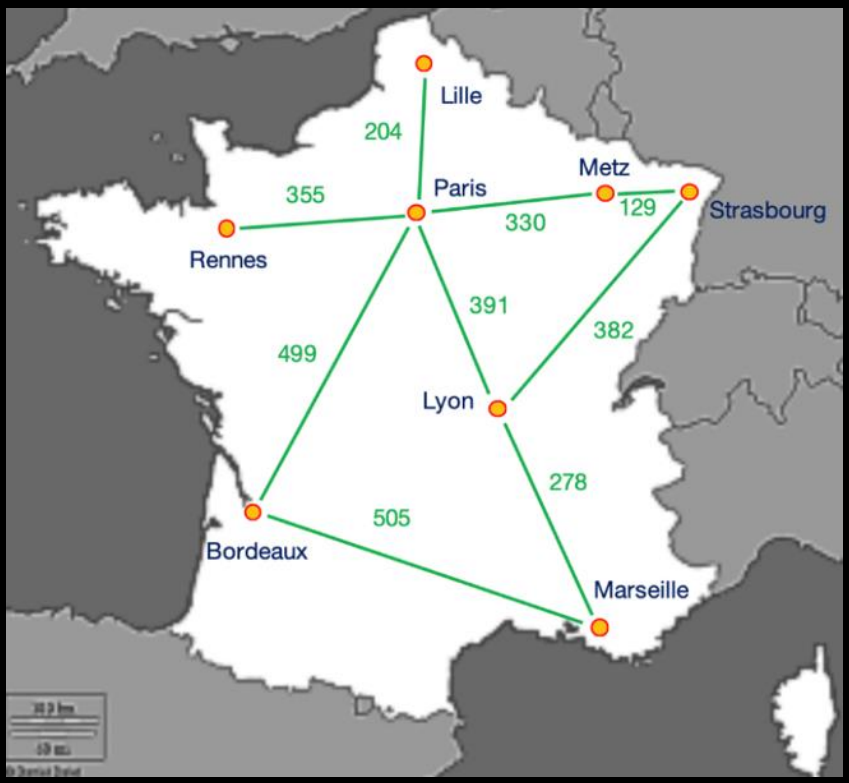
Description :

Travail sur les structures alternatives, itératives et la POO.

Ce mini-projet peut être traité seul ou par un groupe de deux personnes.

Il sera rendu sur MBN.
Le nom du fichier sera du type "*Nom1-Nom2.py*" et les deux noms des élèves devront figurer dans les commentaires de début de programme.

Le but du projet est de trouver le plus court chemin d'un sommet à un autre sommet dans un graphe pondéré, donc d'implémenter l'algorithme de Dijkstra.



On cherche à trouver le plus court chemin entre le sommet *Metz* et le sommet *Bordeaux* en tgv.

Tout au long de l'algorithme, on va garder en mémoire le chemin le plus court depuis le sommet *Bordeaux* pour chacune des autres sommets dans un tableau.

Principe de l'algorithme :

On répète toujours le même processus :

- Choisir le sommet accessible de distance minimale comme sommet à explorer.
- A partir de ce sommet, explorer ses voisins et mettre à jour les distances pour chacun si elle est inférieure à celle que l'on avait auparavant.
- Répéter jusqu'à ce qu'on arrive au sommet d'arrivée ou jusqu'à ce que tous les sommets aient été explorés.

$Étapes$	Met	Str	Par	Ren	Lil	Mar	Lyo	Bor
1	0	∞	∞	∞	∞	∞	∞	∞
2	×	129_{Met}	330_{Met}	∞	∞	∞	∞	∞
3	×	×	330_{Met}	∞	∞	∞	511_{Str}	∞
4	×	×	×	685_{Par}	534_{Par}	∞	511_{Str}	829_{Par}
5	×	×	×	685_{Par}	534_{Par}	789_{Lyo}	×	829_{Par}
6	×	×	×	685_{Par}	×	789_{Lyo}	×	829_{Par}
7	×	×	×	×	×	789_{Lyo}	×	829_{Par}
8	×	×	×	×	×	×	×	829_{Par}

Le plus court chemin est donc $Bor - Par - Met$ et la distance minimale est 829.

Afin d'implémenter l'algorithme de Dijkstra et de mettre en mémoire le tableau ci-dessus, on a besoin de :

- Une liste *visites* qui contient chaque sommet visité ayant eu la distance minimale.
- Un dictionnaire *predecesseurs* qui contient pour chaque sommet visité (clé) son prédécesseur (valeur).
- Un dictionnaire *distances* qui contient pour chaque sommet visité (clé) la distance minimale qui le sépare du départ (valeur).

Évolution de la liste *visites* durant les $étapes\ 1 \longrightarrow 8$:

$[Met] \longrightarrow$
$[Met, Str] \longrightarrow$
$[Met, Str, Par] \longrightarrow$
$[Met, Str, Par, Lyo] \longrightarrow$
$[Met, Str, Par, Lyo, Lil] \longrightarrow$
$[Met, Str, Par, Lyo, Lil, Ren] \longrightarrow$
$[Met, Str, Par, Lyo, Lil, Ren, Mar]$

Évolution du dictionnaire *predecesseurs* durant les étapes $1 \longrightarrow 8$:

$\{Met: Met\} \longrightarrow$
$\{Met: Met, Str: Met\} \longrightarrow$
$\{Met: Met, Str: Met, Par: Met\} \longrightarrow$
$\{Met: Met, Str: Met, Par: Met, Lyo: Str\} \longrightarrow$
$\{Met: Met, Str: Met, Par: Met, Lyo: Str, Lil: Par\} \longrightarrow$
$\{Met: Met, Str: Met, Par: Met, Lyo: Str, Lil: Par, Ren: Par\} \longrightarrow$
$\{Met: Met, Str: Met, Par: Met, Lyo: Str, Lil: Par, Ren: Par, Mar: Lyo\} \longrightarrow$
$\{Met: Met, Str: Met, Par: Met, Lyo: Str, Lil: Par, Ren: Par, Mar: Lyo, Bor: Par\}$

Évolution du dictionnaire *distances* durant les étapes $1 \longrightarrow 8$:

$\{Met: 0\} \longrightarrow$
$\{Met: 0, Str: 129\} \longrightarrow$
$\{Met: 0, Str: 129, Par: 330\} \longrightarrow$
$\{Met: 0, Str: 129, Par: 330, Lyo: 511\} \longrightarrow$
$\{Met: 0, Str: 129, Par: 330, Lyo: 511, Lil: 534\} \longrightarrow$
$\{Met: 0, Str: 129, Par: 330, Lyo: 511, Lil: 534, Ren: 685\} \longrightarrow$
$\{Met: 0, Str: 129, Par: 330, Lyo: 511, Lil: 534, Ren: 685, Mar: 789\} \longrightarrow$
$\{Met: 0, Str: 129, Par: 330, Lyo: 511, Lil: 534, Ren: 685, Mar: 789, Bor: 829\}$

Production et présentation :

Votre travail consiste à implémenter l'algorithme de Dijkstra en POO en utilisant l'exemple du tgv.

Les données sont fournies dans un fichier *.csv*, qui précise les distances entre chaque sommet (ville).

tgv_edges.csv

```
name1;name2;distance
Paris;Lille;204
Lille;Paris;204
Paris;Rennes;335
Rennes;Paris;335
Paris;Bordeaux;499
Bordeaux;Paris;499
Paris;Metz;330
Metz;Paris;330
Paris;Lyon;391
Lyon;Paris;391
Bordeaux;Marseille;505
Marseille;Bordeaux;505
Metz;Strasbourg;129
Strasbourg;Metz;129
Strasbourg;Lyon;382
Lyon;Strasbourg;382
Marseille;Lyon;278
Lyon;Marseille;278
```

Dataframe (*df_edges*)

	name1	name2	distance
0	Paris	Lille	204
1	Lille	Paris	204
2	Paris	Rennes	335
3	Rennes	Paris	335
4	Paris	Bordeaux	499
5	Bordeaux	Paris	499
6	Paris	Metz	330
7	Metz	Paris	330
8	Paris	Lyon	391
9	Lyon	Paris	391
10	Bordeaux	Marseille	505
11	Marseille	Bordeaux	505
12	Metz	Strasbourg	129
13	Strasbourg	Metz	129
14	Strasbourg	Lyon	382
15	Lyon	Strasbourg	382
16	Marseille	Lyon	278
17	Lyon	Marseille	278

Le fichier *tgv_edges.csv* est exploité à l'aide de la bibliothèque *pandas* qui permet de traiter et de structurer des données.
La bibliothèque *pandas* permet aisément de créer des tableaux sous forme de Dataframe grâce à la méthode ***read_csv(fichier,separateur)***.

On navigue dans les dataframes en Python comme en indiquant le nom de la colonne, puis le numéro de la ligne :

df_edges[name2][13] = '*Metz*'

On fournit le programme *dijkstra_eleve.py* qui utilise le dataframe *df_edges* pour créer le graphe.

- Créer une classe **Graphe** ayant comme attributs :
  - *liste_adjacence* : liste d'adjacence représentant le graphe (dic)
  - *distances* : dictionnaire des distances minimales sommet-départ (dic)
  - *predecesseurs* : dictionnaires qui associe à chaque sommet son prédécesseur (dic)
  - *visites* : liste des sommets visités (list)

Attention : On testera chaque méthode par des assertions données avec le fichier *dijkstra_eleve.py*.

- Créer une méthode ***ajout_sommet(sommet,voisin,poids)*** qui ajoute à la *valeur* du dictionnaire *liste_adjacence* le tuple (*voisin*, *poids*) et qui prend en paramètres :
  - *sommet* : sommet étudié (str)
  - *voisin* : voisin du sommet étudié (str)
  - *poids* : poids de l'arc entre le sommet et son voisin, initialisé à 1 (int, float)

  - **Retour** : le dictionnaire *liste_adjacence* de *clé sommet* et de *valeur* une liste de tuple (*voisin*,*poids*) (dic)

Exemple d'une liste d'adjacence du graphe du tgv :

```
{'Paris': [('Lille', 204), ('Rennes', 335), ('Bordeaux', 499), ('Metz', 330), ('Lyon', 391)],
 'Lille': [('Paris', 204)],
 'Rennes': [('Paris', 335)],
 'Bordeaux': [('Paris', 499), ('Marseille', 505)],
 'Metz': [('Paris', 330), ('Strasbourg', 129)],
 'Lyon': [('Paris', 391), ('Strasbourg', 382), ('Marseille', 278)],
 'Marseille': [('Bordeaux', 505), ('Lyon', 278)],
 'Strasbourg': [('Metz', 129), ('Lyon', 382)]}
```

- Créer une méthode ***classement_adjacence( )*** qui trie dans l'ordre alphabétique les *valeurs* (liste des tuples (*voisin*, *poids*)) de la *liste_adjacence* :
  - *None*

  - **Retour** : *liste_adjacence* triée (dic)

Exemple de la liste d'adjacence du graphe du tgv triée :

```
{'Paris': [('Bordeaux', 499), ('Lille', 204), ('Lyon', 391), ('Metz', 330), ('Rennes', 335)],
'Lille': [('Paris', 204)],
'Rennes': [('Paris', 335)],
'Bordeaux': [('Marseille', 505), ('Paris', 499)],
'Metz': [('Paris', 330), ('Strasbourg', 129)],
'Lyon': [('Marseille', 278), ('Paris', 391), ('Strasbourg', 382)],
'Marseille': [('Bordeaux', 505), ('Lyon', 278)],
'Strasbourg': [('Lyon', 382), ('Metz', 129)]}
```

- Créer une méthode ***initialisation(depart)*** qui initialise les dictionnaires *distances* et *predecesseurs* et qui prend en paramètre :
    - *depart* : sommet de départ (str)
    - **Retour** : les dictionnaires *distances* et *predecesseurs* initialisés

Exemple des dictionnaires *distances* et *predecesseurs* initialisés du graphe du tgv avec comme sommet de départ *Metz* :

```
distances == {'Paris': float('inf'),
              'Lille': float('inf'),
              'Rennes': float('inf'),
              'Bordeaux': float('inf'),
              'Metz': 0,
              'Lyon': float('inf'),
              'Marseille': float('inf'),
              'Strasbourg': float('inf')}

predecesseurs == {'Metz': 'Metz'}
```

Remarque : En Python, l'infini $+\infty$ est symbolisé par *float('inf')*.

- Créer une méthode ***chemin(depart,arrivee)*** qui permet de trouver le chemin à partir du dictionnaire des *predecesseurs* et qui prend en paramètre :
    - *depart* : sommet de départ (str)
    - *arrivee* : sommet d'arrivée (str)
    - **Retour** : la liste des sommets pour aller de *depart* à *arrivee* (list)

Exemple de la liste renvoyée par la méthode ***chemin( )*** pour le dictionnaire *predecesseurs* suivant :

```
tgv.predecesseurs = {'Metz': 'Metz',
                     'Paris': 'Metz',
                     'Strasbourg': 'Metz',
                     'Lyon': 'Strasbourg',
                     'Bordeaux': 'Paris',
                     'Lille': 'Paris',
                     'Rennes': 'Paris',
                     'Marseille': 'Lyon'}

tgv.chemin("Metz", "Bordeaux") == ['Metz', 'Paris', 'Bordeaux']
```

- Créer une méthode ***distance_mini( )*** qui renvoie, pour *une étape* donnée, la distance minimale du départ et le sommet qui correspond :
    - *None*
    - **Retour** : la distance minimale du départ et le sommet qui correspond (tuple)

Exemple des valeurs renvoyées par la méthode ***distance_mini ( )*** pour le dictionnaire *distancess* suivant :

```
tgv.distances = {'Paris': 330,
                 'Lille': 534,
                 'Rennes': 665,
                 'Bordeaux': 829,
                 'Metz': 0,
                 'Lyon': 511,
                 'Marseille': 789,
                 'Strasbourg': 129}
dist_mini, ville = tgv.distance_mini()
dist_mini == 829
ville == "Bordeaux"
```

- Créer une méthode ***dijkstra(depart,arrivee)*** qui permet de trouver le chemin et la distance minimale et qui prend en paramètre :
    - *depart* : sommet de départ (str)
    - *arrivee* : sommet d'arrivée (str)
    - **Retour** : la distance minimale du départ et le chemin qui correspond (tuple)

Algorithme :

- Appliquer la méthode ***initialisation( )*** au sommet de départ pour initialiser les dictionnaires *distances* et *predecesseurs*.
- Tant que le sommet n'est pas l'arrivée:
    - Ajouter sommet à la liste *visites*.
    - Pour tous les voisins de distance *dist* de sommet:
        - Si le voisin n'est pas dans *visites*:
            - Si la *distances*[sommet] + *dist* < *distances* [voisin] :
                - *distances*[voisin] = *distances*[sommet] + *dist*
                - *predecesseurs*[voisin] = sommet
    - Appliquer la méthode ***distance_mini( )*** pour rechercher la distance minimale et le sommet correspondant.

Des commentaires et des noms de variables explicites sont attendus. La PEP8 doit être respectée, les docstrings des fonctions renseignées.

Amélioration possible :

- …