Graphes et fichiers

INFORMATIQUE COMMUNE - TP n° 2.9 - Olivier Reynet

A Graphes et fichier CSV

On dispose des données suivantes (cf. figure 1), c'est-à-dire la distance en kilomètres qui séparent les principales villes bretonnes.

Ville	Brest	Quimper	Lorient	Vannes	St-Brieuc	Rennes	St-Malo
Brest	0	70	115	170	145	240	230
Quimper	70	0	70	120	135	215	220
Lorient	115	70	0	60	125	150	185
Vannes	170	120	60	0	115	115	150
Saint-Brieuc	145	135	125	115	0	100	85
Rennes	240	215	150	115	100	0	70
Saint-Malo	230	220	185	150	85	70	0

FIGURE 1 – Distances en km séparant les principales villes bretonnes

Le format CSV (Comma-Separated Values, ou valeurs séparées par des virgules) est l'un des formats les plus anciens et les plus utilisés pour stocker et échanger des données tabulaires. Sa force réside dans sa simplicité : c'est un fichier texte brut, lisible aussi bien par un humain que par n'importe quelle machine.

Dans un fichier CSV, chaque ligne du fichier correspond à une ligne du tableau, et chaque valeur à l'intérieur de cette ligne est séparée par un délimiteur (généralement une virgule) pour représenter les colonnes.

Dans ce TP, nous allons créer et exploiter un fichier `bretagne.csv` (comme indiqué figure 2) contenant les distances routières entre plusieurs villes de Bretagne. Ce fichier est donc structuré ainsi :

- Ligne d'entêtes : `Ville,Rennes,Brest,Quimper,Lorient,Vannes,Saint-Brieuc`
- Lignes suivantes : `Rennes,0.0,244.0,215.0,152.0,113.0,99.0`

B Maniupulation d'un fichier CSV

En Python, l'interaction avec un fichier suit toujours un cycle de vie en trois étapes : **ouverture**, **opération** (lecture ou écriture) et **fermeture**.

```
Ville,Brest,Quimper,Lorient,Vannes,St-Brieuc,Rennes,St-Malo
Brest,0,70,115,170,145,240,230
Quimper,70,0,70,120,135,215,220
Lorient,115,70,0,60,125,150,185
Vannes,170,120,60,0,115,115,150
Saint-Brieuc,145,135,125,115,0,100,85
Rennes,240,215,150,115,100,0,70
Saint-Malo,230,220,185,150,85,70,0
```

FIGURE 2 – Contenu du fichier CSV `bretagne.csv`

Ouvrir et fermer un fichier

`open(chemin, mode)`

Cette fonction est le point d'entrée. Elle demande au système d'exploitation d'accéder à un fichier et renvoie un ń objet fichier ż (ou *file handler*) qui permet d'interagir avec lui. Le paramètre `mode` définit ce que vous comptez faire :

- `'r'` (read) : lecture seule autorisée (mode par défaut).
- `'w'` (write) : écriture autorisée. **Attention :** si le fichier existe, son contenu est effacé. S'il n'existe pas, il est créé.
- `'a'` (append) : ajouter à la fin du fichier existant. Si le fichier existe, son contenu n'est pas effacé. S'il n'existe pas, il est créé.

`close()`

Une fois les opérations terminées, il est crucial d'appeler cette méthode sur l'objet fichier. Elle indique au système d'exploitation que vous avez fini, forçant l'enregistrement des données résiduelles en mémoire et libérant les ressources système. Cela permet à d'autres utilisateurs d'utiliser ce fichier : en effet, lorsqu'on ouvre un fichier en écriture, le système bloque l'écriture à tous les autres utilisateurs. Cela permet de préserver la cohérence et l'intégrité des fichiers.

```
fichier = open("mon_fichier.csv","r")  # ouverture du descripteur de fichier
# faire quelque chose avec ce fichier
fichier.close()  # fermeture du descripteur de fichier
```

R La bonne pratique est d'utiliser le gestionnaire de contexte, c'est-à-dire l'instruction `with`. Elle gère automatiquement l'appel à `close()` à la fin du bloc, même si une erreur survient pendant l'exécution.

```
with open("mon_fichier.txt", "rwa") as fichier:
    # Operations de lecture/ecriture ici
# Le fichier est automatiquement fermé ici
```

Lire le contenu : mode `'r'`

`readline()`

Cette méthode lit et renvoie **une seule ligne** du fichier à la fois, en incluant le caractère de saut de ligne (`"\n"`) à la fin. À chaque appel successif, elle passe à la ligne suivante.

R C'est la méthode idéale dans une boucle `while` ou `for` pour lire de très gros fichiers sans saturer la mémoire de l'ordinateur. On lit au fur et à mesure. Voici quelques exemples d'utilisation :

```
fichier = open("mon_fichier.csv","r")
for ligne in fichier: # on a même pas besoin de readline(), Python gère tout seul !
    print(ligne)
fichier.close()
```

```
with open("mon_fichier.csv", "r") as fichier:
    numero_ligne = 1
    ligne_actuelle = fichier.readline()
    # Tant que la ligne lue n'est pas vide (donc tant qu'on n'est pas à la fin)
    while ligne_actuelle:
        print(numero_ligne," - ", ligne_actuelle)
        # On passe à la ligne suivante
        ligne_actuelle = fichier.readline()
        numero_ligne += 1
```

`readlines()`

Cette méthode lit tout le reste du fichier d'un seul coup et renvoie **une liste de chaînes de caractères**, où chaque élément de la liste correspond à une ligne du fichier (incluant toujours le `\n`).

R C'est une fonction très pratique pour les petits fichiers où l'on souhaite manipuler, trier ou filtrer l'ensemble des lignes directement en mémoire.

```
with open("mon_fichier.csv", "r") as fichier:
    # On récupère toutes les lignes sous forme de liste
    liste_des_lignes = fichier.readlines()

print(liste_des_lignes)
```

Écrire du contenu : modes `'w'` ou `'a'`)

`write(chaine)`

Cette méthode insère la chaîne de caractères fournie directement dans le fichier.

R Contrairement à la fonction `print()`, `write()`, la fonction `write` **n'ajoute pas de saut de ligne automatiquement**. Si vous voulez écrire sur plusieurs lignes, vous devez explicitement ajouter le caractère `"\n"` à la fin de votre chaîne (ex : `fichier.write("Bonjour\n")`).

`writelines(iterable)`

Cette méthode prend un itérable (comme une liste ou un tuple de chaînes de caractères) et écrit tous ses éléments dans le fichier à la suite.

R Le nom de cette méthode est un peu trompeur. Tout comme `write()`, elle **n'ajoute aucun saut de ligne** entre les éléments. Si votre liste est `["A", "B", "C"]`, le fichier contiendra `ABC`. Pour avoir des lignes séparées, les éléments de la liste doivent déjà contenir leurs propres `\n` (ex : `["A\n", "B\n", "C\n"]`).

C Écriture d'un fichier CSV

Le code du script Python qui crée le fichier csv à partir des données de la Bretagne ainsi que le fichier `bretagne.csv` est donné (cf. site). Lire le code et essayer de le comprendre.

D Lecture d'un fichier CSV

D1. Écrire une fonction `lire_csv_v1(nom_fichier)` qui ouvre le fichier en utilisant les primitives `open` et `close`, lit toutes les lignes, et renvoie une liste contenant les lignes brutes (chaînes de caractères). Tester sur le fichier `bretagne.csv`

D2. En Python, il est recommandé d'utiliser un gestionnaire de contexte pour s'assurer de la fermeture des fichiers. Écrire une fonction `lire_csv(nom_fichier)` qui utilise la syntaxe `with open(...)as f :` pour lire le fichier. Cette fonction devra extraire les données, séparer les colonnes (avec la méthode `split`) et renvoyer deux éléments :

1. une liste `villes` contenant le nom des villes (chaînes de caractères),
2. et une matrice `matrice_dist` (liste de listes) contenant les distances converties en flottants (`float`).

E Création d'un graphe pondéré

Nous allons modéliser notre carte de Bretagne sous la forme d'un graphe pondéré, représenté par des listes d'adjacence. Les sommets sont numérotés de 0 à $n-1$. Le graphe sera stocké dans une liste `g` où `g[u]` est une liste de couples `(v, poids)` signifiant qu'il existe une arête de poids `poids` entre le sommet `u` et le sommet `v`.

E3. Écrire une fonction de signature `creer_graphe(matrice_dist: list[list[float]])` qui prend en paramètre la matrice des distances et renvoie la liste d'adjacence qui correspond au graphe des villes pondéré par les distances. On ne créera pas d'arête si la distance est nulle (distance d'une ville à elle-même).

F Visualisation du graphe

Le code suivant utilise la bibliothèque `networkx` et permet de construire une visualisation du graphe des distances entre les villes.

```python
import networkx as nx
import matplotlib.pyplot as plt

def tracer_graphe(adj, villes):
    G = nx.Graph()
    n = len(villes)

    # Ajout des œnuds avec le nom de la ville en attribut
    for i in range(n):
        G.add_node(i, nom=villes[i])

    # Ajout des arêtes
    for u in range(n):
        for voisin in adj[u]:
            v = voisin[0]
            poids = voisin[1]
            if u < v:
                G.add_edge(u, v, weight=poids)

    # Dessin
    pos = nx.spring_layout(G)  # Positionnement des sommets
    labels = {}
    for i in range(n):
        labels[i] = villes[i]

    nx.draw(G, pos, labels=labels, with_labels=True, node_color="lightblue",
        node_size=2000)

    labels_aretes = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels_aretes)
    plt.show()
```

G Voyageur de commerce

Le problème du voyageur de commerce (ou *Travelling Salesperson Problem*, TSP) consiste à trouver le plus court chemin passant par un ensemble de villes une et une seule fois, et revenant à la ville de départ, c'est-à-dire trouver un cycle le plus court possible. C'est un problème d'optimisation combinatoire classique.

Évaluation d'un cycle

Un *cycle* est représenté par une liste d'entiers correspondant aux indices des villes visitées, dans l'ordre. Par exemple, le cycle `[0, 1, 3, 2]` signifie que le voyageur part de la ville 0, visite la 1, puis la 3, puis la 2, avant de revenir à la ville 0.

G4. Écrire une fonction de signature `cout_cycle(matrice_dist, cycle)` qui prend en paramètre la matrice des distances et un cycle (liste d'indices), et qui renvoie la distance totale parcourue par le voyageur (incluant le retour au point de départ).

Heuristique gloutonne : choix du plus proche voisin

La recherche exacte de la solution optimale peut être extrêmement longue pour un grand nombre de villes. On commence par implémenter une heuristique gloutonne : à chaque étape, le voyageur choisit de se rendre dans la ville **non visitée la plus proche de sa position actuelle**.

G5. Écrire une fonction de signature `plus_proche_voisin(matrice_dist, depart)` qui renvoie un tuple `(cycle, cout_total)`. Cette fonction doit construire le cycle en appliquant la stratégie gloutonne à partir de l'indice de la ville de `depart`. Vous pouvez initialiser la recherche de minimum avec `math.inf`.

G6. L'algorithme précédent dépend fortement de la ville de départ. Écrire une fonction `meilleur_glouton(matrice_dist)` qui simule l'algorithme glouton en partant de *chaque* ville possible, et renvoie le meilleur cycle trouvé ainsi que son coût minimal.