TD

Python

Voyageur de commerce

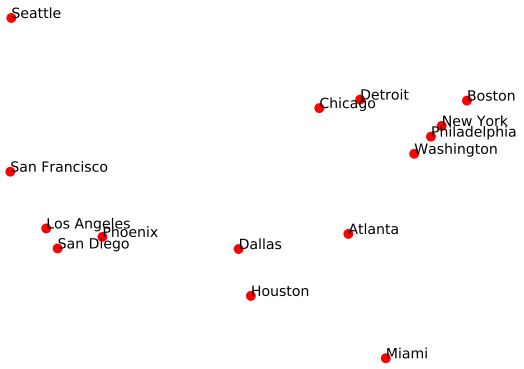
1. Présentation

Dans le problème du voyageur de commerce, un représentant commercial doit visiter n villes toutes reliées entre elles directement par une ligne aérienne (rectiligne). On dispose donc d'un graphe complet de villes où chaque sommet (les villes) possède $n-1$ voisins. Le problème du voyageur consiste à calculer la tournée de longueur minimale visitant une et une seule fois chaque ville. La longueur de la tournée inclut le retour au point de départ.
Les n villes $(V_n)_{n\in[\![0,N-1]\!]}$ sont repérées par leurs coordonnées dans le plan euclidien $(x_i, y_i)_{i\in[\![0,n-1]\!]}$.

Une tournée correspond à un chemin particulier qui passe par ces villes. Pour changer de tournée, il suffit de permuter l'ordre des villes.

On dispose du dictionnaire `dico_xy` dont les clés sont les noms des villes et les valeurs correspondent aux couples (x, y) des coordonnées de la villes. Ce dictionnaire est une variable globale utilisée dans la suite du problème. Par exemple pour les principales villes américaines le script ci-dessous permet de visualiser leur position. Ce script présente également les seules importations autorisées.

```
1  from random import random
2  import matplotlib.pyplot as plt
3  from math import sqrt, exp
4
5  villes = list(dico_xy.keys())
6  xlist, ylist = [], []
7  for ville, (x,y) in dico_xy.items():
8      xlist.append(x)
9      ylist.append(y)
10     plt.text(x, y, ville)
11 plt.plot(xlist, ylist, 'ro')
```



Exercice 1 Une tournée est codée par la liste `tournee` contenant les noms des villes (type `str`) dans l'ordre emprunté par le voyageur de commerce. Rédiger une procédure `trajet(tournee)` qui permet de représenter graphiquement une tournée (en finissant sur la ville de départ).

Exercice 2 Proposer une fonction `dico_adj(dico_xy)` qui renvoie le dictionnaire d'adjacence du graphe de villes à partir du dictionnaire `dico_xy`. Le dictionnaire d'adjacence possède des clés qui sont les n noms des villes. La valeur associée à une clé (par exemple `'Boston'`) est un dictionnaire dont les clés sont les $n-1$ autres villes et les valeurs les distances entre ces villes et la ville associée à ce dictionnaire (par exemple `'Boston'`).

Dans le cas où l'on ne considère que les 5 plus grandes villes américaines le dictionnaire d'adjacence est

```
{'New York': {'Los Angeles': 44, 'Chicago': 13, 'Miami': 16, 'Dallas': 24},
 'Los Angeles': {'New York': 44, 'Chicago': 31, 'Miami': 39, 'Dallas': 21},
 'Chicago': {'New York': 13, 'Los Angeles': 31, 'Miami': 17, 'Dallas': 12},
 'Miami': {'New York': 16, 'Los Angeles': 39, 'Chicago': 17, 'Dallas': 17},
 'Dallas': {'New York': 24, 'Los Angeles': 21, 'Chicago': 12, 'Miami': 17}}
```

Une unité de longueur correspond environ à 100 km.

Exercice 3 Indiquer la distance totale parcourue (on parle alors de coût) pour la tournée suivante : `['New York', 'Miami', 'Chicago', 'Los Angeles', 'Dallas']`, sans oublier de revenir à la ville de départ.
Proposer une fonction `cout_voyage(graph,tournee)` qui renvoie cette distance pour le graphe et la tournée précisés en argument.

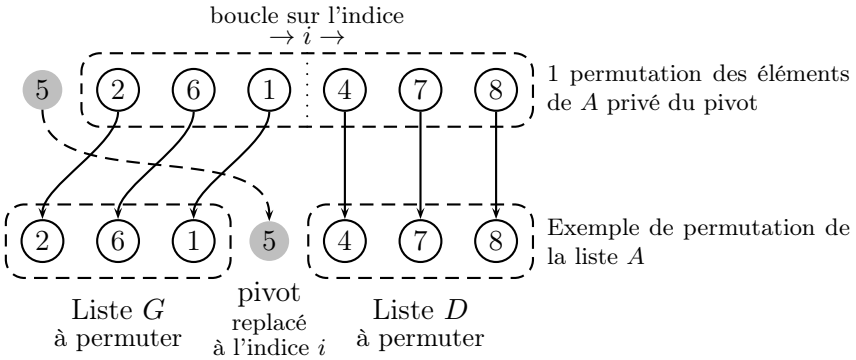
2. Méthode force brute

Le but de cette section est de réaliser un algorithme naïf testant toutes les tournées possibles pour trouver le coût minimal. Il s'agit donc de trouver toutes les permutations d'une liste `tournee` codant la tournée. On s'inspire de l'algorithme suivant réalisant toutes les permutations de la liste A de taille n :

$$A = \boxed{5}\boxed{2}\boxed{6}\boxed{1}\boxed{4}\boxed{7}\boxed{8}$$

Le pivot est choisi comme étant le premier élément de la liste.

$Pivot \leftarrow 5$
Pour l'ensemble des permutations de A privé du pivot
Pour chaque indice i de 0 à $n-1$
Liste de gauche : les éléments d'indice inférieur à i (par exemple pour $i = 3$) : $G = \boxed{2}\boxed{6}\boxed{1}$
Liste de droite : les éléments d'indice supérieur à i : $D = \boxed{4}\boxed{7}\boxed{8}$



Exercice 4 Proposer une fonction récursive `ensemble_permutation(tournee:list) -> list` qui à partir de la liste d'une tournée renvoie une liste contenant l'ensemble des listes de permutation de la tournée.

Exercice 5 En appelant C_n la complexité de la fonction précédente pour une liste de n éléments, indiquer la relation de récurrence vérifiée par C_n. En déduire la complexité de cette fonction. En admettant qu'une permutation s'obtienne en 1 ns, combien de siècles faudra-t-il pour obtenir

toutes les permutations d'une liste de 21 éléments ?

Exercice 6 En déduire une fonction `voyage(graph)` qui prend en argument un graphe de villes (c'est-à-dire le dictionnaire d'adjacence associé) et qui renvoie la tournée la plus courte et son coût.

3. Avec une heuristique

Comme le problème du voyageur de commerce est difficile, on considère l'heuristique simple qui construit une tournée en partant d'une ville de départ et qui se dirige vers la ville à la plus courte distance qui n'a pas encore été visitée.

Exercice 7 Rédiger une fonction `plusproche(graph, ville, visitees)` qui prend comme argument le dictionnaire d'adjacence du graphe de villes (`graph`), une `ville` appartenant au graphe et la liste des villes déjà visitée (`visitee`. Cette fonction renvoie la ville la plus proche (n'appartenant pas aux villes déjà visitées) et la distance les séparant.
Proposer un code présentant une complexité optimale.

Exercice 8 En déduire une fonction `voyage2(graph)` qui prend en argument un graphe de villes (c'est-à-dire le dictionnaire d'adjacence associé) et qui renvoie la tournée la plus courte et son coût.

Exercice 9 À quelle famille d'algorithme appartient cette heuristique ? Préciser la complexité de la fonction proposée.

4. Utilisation de l'algorithme de Prim

4.1. Définitions

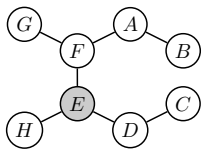
Définition

Un **arbre** est un graphe connexe non orienté, sans cycle, où des nœuds sont reliés par des arêtes.

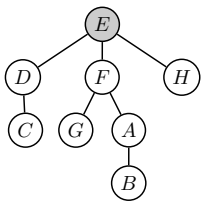
Contrairement aux graphes, l'organisation des nœuds d'un arbre comporte une dimension hiérarchique.

Définition

Lorsqu'un sommet est distingué parmi les autres, il est nommé **racine** et la structure d'arbre se nomme alors **arborescence** associée à cette racine.
La représentation graphique d'une arborescence est faite de façon analogue à celle d'un arbre en botanique mais à l'envers avec la racine en haut pour garder l'idée de hiérarchie.



⇒ Représentation de l'arborescence depuis la racine E



⚠ Par abus de langage dans tout le reste du sujet, on confond arbre et arborescence.

4.2. Présentation de l'algorithme

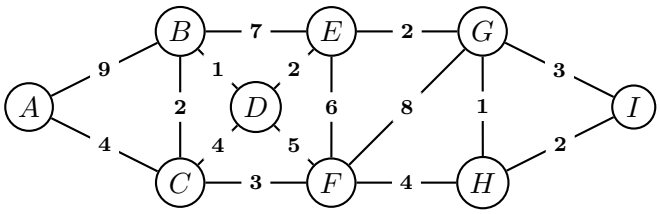
L'algorithme de Prim calcule un arbre couvrant minimal dans un graphe connexe pondéré et non orienté. Il permet de déterminer un sous-ensemble d'arêtes formant un arbre sur l'ensemble des sommets du graphe initial tel que la somme des poids de ses arêtes soit minimale.

```
from queue import PriorityQueue

def prim(graph, depart):
    cout = {}
    predecesseur = {}
    for sommet in graph:
        cout[sommet] = float('inf')
    cout[depart] = 0
    prior = PriorityQueue()
    prior.put((0, depart))
    visite = []
    while not prior.empty():
        sommet = prior.get()[1]
        visite.append(sommet)
        for voisin, poids in graph[sommet].items():
            distance = poids
            if voisin not in visite and distance < cout[voisin]:
                cout[voisin] = distance
                prior.put((cout[voisin], voisin))
                predecesseur[voisin] = sommet
    return predecesseur
```

Le code ci-dessus donne un exemple d'implémentation de l'algorithme utilisant une file de priorité. La fonction `prim` renvoie le dictionnaire `successeur` dont les valeurs sont les prédécesseurs des clés.

Exercice 10 Représenter l'arbre couvrant minimal obtenu selon l'algorithme de Prim pour le graphe pondéré connexe $\mathcal{G}_0$ ci-dessous en partant du sommet A.
Préciser l'ordre de visite des différents sommets.



Exemple de graphe $\mathcal{G}_0$

Exercice 11 L'algorithme de Prim est similaire à celui de Dijkstra, il suffit de modifier une seule ligne pour obtenir les plus courts chemins depuis `depart` vers tous les autres sommets.

1) Indiquer la modification à effectuer.

2) Préciser le plus court chemin entre A et I.

4.3. Cas du voyageur de commerce

Un algorithme approché pour la résolution du problème du voyageur de commerce consiste d'abord à déterminer un arbre couvrant minimal $\mathcal{T}$. La tournée est ensuite définie par l'ordre de visite des

sommets selon un parcours en profondeur de l'arbre $\mathcal{T}$

Exercice 12 Proposer une fonction récursive `dfs(graph, depart, liste=None)` qui prend un sommet `depart` en argument et renvoie la liste `liste` des sommets du graphe listés dans l'ordre d'un parcours en profondeur à partir du sommet `depart`.

Exercice 13 Appliquer le parcours en profondeur sur l'arbre couvrant de racine A du graphe $\mathcal{G}_0$.