Correction

Python

Voyageur de commerce

1. Présentation

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).

```
def trajet(tournee, option='k:'):
    n = len(tournee)
    for i in range(n):
        x, y = dico_xy[tournee[i]]
        xv, yv = dico_xy[tournee[(i+1)%n]]
        plt.plot([x, xv], [y, yv], option)
```

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'`).

```
def dico_adj(dico_xy):
    villes = list(dico_xy)
    graph = {}
    for ville in villes:
        graph[ville] = {}
        for voisine in villes:
            if voisine != ville:
                x, y = dico_xy[ville]
                xv, yv = dico_xy[voisine]
                graph[ville][voisine] = int(sqrt((x-xv)**2 + (y-yv)**2))
    return graph
```

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.

La distance totale parcourue pour la tournée `['New York', 'Miami', 'Chicago', 'Los Angeles', 'Dallas']`, (sans oublier de revenir à la ville de départ) est : $16+17+31+21+24=109$

Une fonction `cout_voyage(graph,tournee)` qui renvoie cette distance :

```
def cout_voyage(graph, tournee):
    n = len(tournee)
```

```
    somme = 0
    for i in range(n):
        ville = tournee[i]
        voisin = tournee[(i+1)%n]
        somme += graph[ville][voisin]
    return somme
```

2. Méthode force brute

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.

```
def ensemble_permutation(tournee):
    n = len(tournee)
    if n == 1:
        return [tournee]
    ensemble = []
    pivot = tournee[0]
    for permutation in ensemble_permutation(tournee[1:]):
        for pos in range(n):  # les for peuvent être inversés
            gauche = permutation[:pos]
            droite = permutation[pos:]
            ensemble.append(gauche + [pivot] + droite)
    return ensemble
```

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 ?

En notant C_n la complexité de la fonction pour une liste de taille n, la relation de récurrence vérifiée par C_n est $\boxed{C_n = C_{n-1} + n^2(n-1)!}$. En effet on appelle la fonction sur une liste réduite d'un élément puis on effectue 1 boucle (`for permut...`) sur le nombre de permutation d'une liste de taille $n-1$ soit $(n-1)!$ tours, pour chaque tour on effectue n boucles (`for pos`) et encore environ n opérations avec le sclicing et la concaténation.

On en déduit (en sommant)

$$C_n = \sum_{k=1}^{n} k^2(k-1)! = \sum_{k=1}^{n} k \cdot k! = \sum_{k=1}^{n} (k+1-1) \cdot k! = \sum_{k=1}^{n} (k+1)! - \sum_{k=1}^{n} (k)!$$

La somme télescopique donne finalement :

$$C_n = (n+1)! - 1$$

D'où la complexité factorielle $\mathcal{O}((n+1)!)$.
En admettant qu'une permutation s'obtienne en 1 ns, il faudra 22! ns soit environ 350 siècles 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.

```python
def voyage(graph):
    villes = list(graph.keys())
    mini = float('inf')
    for tournee in ensemble_permutation(villes):
        cout = cout_voyage(graph, tournee)
        if cout < mini:
            mini = cout
            mini_tournee = tournee
    return mini_tournee, mini
```

3. Avec une heuristique

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.

```python
def plusproche(graph, ville, visitees):
    mini = float('inf')
    for voisin, dist in graph[ville].items():
        if dist < mini and voisin not in visitees: # complexité moins bonne
            mini = dist
            min_voisin = voisin
    return min_voisin, mini
```

```python
def plusproche(graph, ville, visitees):
    mini = float('inf')
    villes_non_visit = [v for v in list(graph.keys()) if v not in visitees]
    for voisin in villes_non_visit:
        dist = graph[ville][voisin]
        if dist < mini:
            mini = dist
            min_voisin = voisin
    return min_voisin, mini
```

La complexité est linéaire (on peut pas faire mieux).

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.

```python
def voyage2(graph):
    n = len(graph)
    villes = list(graph.keys())
    depart = villes[0]
    tournee = [depart]
    somme = 0
    for boucle in range(n-1):
        voisin, dist = plusproche(graph, depart, tournee)
        tournee.append(voisin)
        depart = voisin
        somme += dist
    somme += graph[voisin][villes[0]] # relier le depart
```

```
13      return tournee, somme
```

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

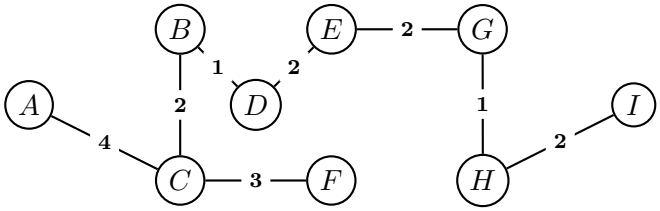
Il s'agit d'un algorithme glouton qui s'attache à prendre la ville localement la plus proche sans se soucier du parcours global.
La complexité est quadratique $\mathcal{O}(n^2)$ car on effectue $n-1$ appels d'une fonction linéaire.

4. Utilisation de l'algorithme de Prim

4.1. Définitions

4.2. Présentation de l'algorithme

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 d'arbre couvrant depuis A

L'ordre de visite est A, C, B, D, E, G, H, I, F.

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.

La modification :

```
16              distance = cout[sommet] + poids
```

Le plus court chemin de A à I est :

$$A \xrightarrow{4} C \xrightarrow{7} F \xrightarrow{11} H \xrightarrow{13} I$$

Cette distance est plus courte que celle obtenue par Prim (14).

4.3. Cas du voyageur de commerce

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`.

```python
def dfs(graph, depart, parcours=None):
    if parcours is None:
        parcours = []
    parcours.append(depart)
    for  voisin in graph[depart]:
        if voisin not in parcours:
            dfs(graph, voisin, parcours)
    return parcours
```

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

Le parcours est :

$$A \to C \to B \to D \to E \to G \to H \to I \to F$$