TP : Problème du voyageur de commerce, recuit simulé

1 Introduction

On s'intéresse au problème du *voyageur de commerce* dans lequel un voyageur doit visiter N villes une seule fois puis revenir à son point de départ. On cherche à déterminer le chemin le plus court possible.

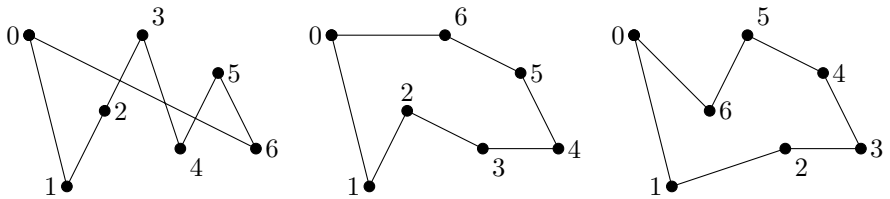

FIG. 1 : Quelques exemples de chemins que pourrait suivre le voyageur de commerce dans un problème à 7 villes. Les numéros indiquent l'ordre de passage dans chaque ville.

La difficulté de ce problème tient au fait qu'il devient très vite impossible de tester tous les trajets possibles lorsque le nombre de villes augmente. En effet, il existe $\frac{(N-1)!}{2}$ parcours possibles pour N villes (car on peut fixer arbitrairement la ville de départ et chaque chemin peut être parcouru dans les deux sens).

Dans le tableau ci-dessous on donne le nombre de chemins possibles pour différents nombres de villes ainsi que le temps de calcul nécessaire pour tester toutes les possibilités en comptant que l'évaluation d'un chemin prend 1 ns, ce qui même avec des ordinateurs récents est extrêmement faible.

Nombre de villes	Nombre de parcours possibles	temps de calcul
10	2×10^5	0.1 ms
15	4×10^{10}	43 s
20	6×10^{16}	2 ans
25	3×10^{23}	10 millions d'années
30	4×10^{30}	10^{14} années

Nous devons donc oublier l'examen systématique de tous les chemins possibles et trouver une autre méthode pour déterminer le chemin le plus court. La méthode utilisée dans ce TP est celle du *recuit simulé*.

2 Principe de la méthode

La méthode du recuit simulé s'inspire de la métallurgie où il est parfois nécessaire de recuire des pièces métalliques pour en changer les propriétés physiques. En effet, lorsqu'on déforme un morceau de métal, on crée des défauts dans le réseau cristallin qui ont pour effet de rendre la pièce plus cassante. Pour restaurer la plasticité de la pièce métallique, on la porte à haute température pour augmenter la mobilité des atomes du cristal puis on la refroidit lentement en laissant les atomes reprendre leur position dans le réseau cristallin. Le chauffage du métal fournit aux atomes l'énergie nécessaire pour s'extraire d'un minimum local d'énergie pour se retrouver dans un minimum global (réseau cristallin parfait).

Pour déterminer le chemin le plus court dans le problème du voyageur de commerce, on peut commencer par un chemin quelconque puis on lui apporte des modifications aléatoires en ne conservant que celles qui induisent une diminution de la longueur du chemin. Cependant en procédant de cette manière on se retrouve très vite dans un *minimum local*, c'est à dire un chemin dont la longueur ne peut plus être réduite par un seul petit changement. Dans ce cas, ou doit être amené à *augmenter la température*, c'est-à-dire, à accepter une modification qui augmente la longueur du chemin en espérant que les modifications suivantes permettront de la réduire davantage.

On représente schématiquement cette situation sur la figure ci-dessous :

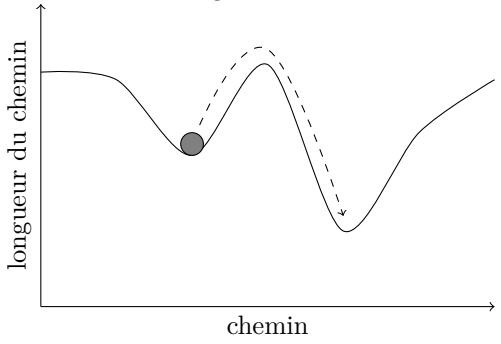

FIG. 2 : Situation où l'on a atteint un minimum local et où il est nécessaire d'accepter une augmentation de la longueur du chemin pour espérer atteindre le minimum global

3 Fonctions utiles

Nous allons essayer de trouver une solution au problème du voyageur de commerce pour un ensemble de villes de France. Le fichier `villes.txt` contient 58 lignes de la forme :

`Nom de la ville, latitude, longitude, population`

1. Compléter la fonction `chargerVilles()` suivante qui lit le fichier et remplit une liste stockée dans une variable globale `VILLES` dont chaque élément contient le nom, les coordonnées et la population d'une ville. On pourra utiliser la fonction `split` qui permet de séparer une chaine de caractères en une liste. Par exemple si la variable `a` contient la chaine `"a,b,c"`, alors `a.split(',')` renvoie la liste `["a","b","c"]`. On prendra soin de convertir les coordonnées en `float`.

```
VILLES = []

def chargerVilles():
    global VILLES  # La variable VILLES est une variable globale
    fichier = open("Nom du fichier", "r")
    for l in fichier:
        # À COMPLÉTER
    fichier.close()
```

Un parcours sera représenté par une liste d'entiers représentant le numéro de la ville visitée. Par exemple le parcours `c = [1, 0, 5, 4]` représente un chemin qui passe d'abord par la ville 1 de la liste `VILLES` puis par la ville 0, …, pour finir par revenir à la ville 1 de départ. Le retour à la ville de départ est implicite et n'apparaîtra pas dans la liste.

2. Compléter la fonction suivante qui affiche un parcours sur un graphique.

```
# Bloc à placer au début du programme
import matplotlib.pyplot as plt
# Fin du bloc à placer au début du programme

def afficheCircuit(c):
    # À COMPLÉTER
    # stocker dans les listes x et y les coordonnées (longitude et latitude)
    # des villes visitées
    ax.plot(x,y,'k-o') # affiche les villes
    plt.show()         # affiche le graphique
```

3. La distance entre deux points P_1 et P_2 de la surface de la Terre définis par leurs longitudes (Ψ_1 et Ψ_2) et latitudes (φ_1 et φ_2) est :

$$d = R \arccos\left(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(\Psi_1 - \Psi_2)\right)$$

Où $R = 6370.7\,\text{km}$ est le rayon de la Terre.

Écrire une fonction `distance(P1, P2)` qui prend en paramètre deux points et qui renvoie la distance qui les sépare. `P1` et `P2` sont des listes de la forme `[latitude, longitude]`.

On pourra importer la bibliothèque `math` qui fournit la fonction `math.acos(x)` qui renvoie la valeur de $\arccos(x)$ en radians.

4. Écrire une fonction `distanceCircuit(c)` qui prend en paramètre un chemin et qui renvoie la distance totale de ce chemin. On n'oubliera pas que le chemin doit revenir à son point de départ.

4 Force brute

Il est temps de passer à la recherche d'un chemin court ! Nous allons commencer par essayer la force brute et tester tous les chemins possibles ! On ne s'attend pas à pouvoir l'appliquer au 58 villes mais juste à un petit nombre. Pour énumérer toutes les permutations possibles des villes nous allons utiliser un algorithme récursif qui utilise l'idée suivante. Les permutations des éléments d'une liste `L` de longueur n sont obtenues en prenant les n valeurs possibles de `L[-1]` précédées de toutes les permutations possibles des autres éléments de `L`.

Le programme suivant affiche à l'écran la liste des permutations des éléments d'une liste `L` en suivant cette idée.

```
# Affiche les valeurs possibles de L en permutant ses k premiers éléments
def permutations(k, L):
    # Si on ne veut permuter que le premier élément il n'y a plus rien à faire
    if k==1:
        print(L)
```

```
    else:
        # Sinon on permute les k-1 premiers éléments pour toutes
        # les valeurs possibles du k-ième élément
        for i in range(k):
            # Place le ième élément en position k-1
            L[k-1], L[i] = L[i], L[k-1]
            # Trouve les permutations correspondantes
            permutations(k-1, L)
            # Remet l'élément à sa place
            L[k-1], L[i] = L[i], L[k-1]
```

5. Modifier la fonction précédente pour écrire une fonction qui teste tous les chemins possibles du voyageur de commerce et renvoie le chemin le plus court. On pourra utiliser une variable globale pour stocker le meilleur chemin exploré. On pourra tester son fonctionnement sur un nombre restreint de villes, par exemple sur les n premières de la liste `VILLES`.

On se rend vite compte que cette méthode n'est pas utilisable pour trouver le meilleur chemin reliant plus de quelques villes. Il est temps de passer à la méthode du recuit simulé.

5 Recuit simulé

Une première idée pour trouver un chemin le plus court possible consiste à faire subir des modifications aléatoires à un chemin de départ et à ne conserver que celles qui en réduisent la longueur. Le premier type de modifications que nous allons considérer est l'échange de deux villes dans le circuit du voyageur. Cependant on a vite fait de se trouver avec un chemin dont la longueur ne peut plus être réduite par un échange mais qui pourrait l'être par plusieurs échanges consécutifs. Il faut alors accepter de procéder à un échange qui augmente la longueur du chemin pour espérer le réduire par la suite.

L'algorithme utilisé est le suivant :

initialiser le chemin C à une valeur quelconque
pour $i = 0$ **à** N_{tot} **faire**
  T = Température calculée à partir des valeurs de i et N_{tot}
  d_1 = distanceCircuit(C)
  Échanger deux villes aléatoires de C
  d_2 =distanceCircuit(C)
  si $d_2 < d_1$ **alors**
    Conserver la modification
  sinon
    Conserver la modification avec une probabilité $e^{-(d_2-d_1)/T}$
  fin si
fin pour
Renvoyer le chemin C et sa distance

Indication : Pour exécuter une portion de programme avec une probabilité p, on peut utiliser le code suivant :

```
import random
if random.random()<p:
  # Partie à exécuter avec la probabilité p
```

6. On souhaite que la température diminue progressivement (linéairement ou autre, à vous d'expérimenter) au cours du temps pour passer de `Tmax` au début de la simulation à `0` à la fin. Écrire une fonction d'entête

```
calculeTemperature(i:int, N:int) -> float
```

qui renvoie la température calculée pour l'itération numéro `i` sur `N`. `Tmax` est une variable globale dont vous pourrez déterminer une valeur qui fonctionne bien.

7. Écrire une fonction `echange(C,i,j)` qui échange les villes d'indices `i` et `j` du circuit `C`.

8. Écrire une fonction d'entête

```
simulation(N:int) -> (list, float)
```

qui effectue une simulation de recuit pour aider le voyageur de commerce à visiter toutes les villes par le chemin le plus court possible. La fonction renverra le chemin trouvé et sa distance.

Vous pouvez maintenant utiliser la fonction `simulation` pour essayer de trouver un chemin le plus court possible. En affichant ces chemins sur une carte, vous pourrez essayer d'imaginer une ou plusieurs méthodes pour améliorer la solution obtenue. On pourra aussi écrire une fonction qui effectue une série de simulations et qui n'en conserve que le meilleur résultat.

6 Améliorations de la solution

6.1 Minimum local

Les simulations de la partie précédente ont du vous permettre de déterminer des chemins *pas trop longs* passant par toutes les villes. En examinant les solutions obtenues vous pourrez vous rendre compte que certaines sont évidemment perfectibles.

La première chose à faire est de s'assurer qu'il n'existe plus d'échange de villes qui puisse améliorer la solution.

9. Écrire une fonction d'entête

```
optimisationEchange(c:list) -> (list,float)
```

qui prend en paramètre un circuit `c` et qui teste tous les échanges de villes possibles en ne conservant que ceux qui diminuent la longueur du parcours. Votre fonction renverra le chemin atteint ainsi que sa longueur lorsqu'il n'existe plus d'inversion qui puisse réduire la longueur du chemin. Cela assure que l'on a atteint un minimum local de la longueur du chemin.

6.2 Algorithme 2-opt

En cherchant un chemin de distance minimale avec la méthode utilisée jusqu'à maintenant on trouve des chemins qui ne sont manifestement pas les meilleurs possibles, on trouve notamment des chemins dans lesquels des segments se croisent. Or il est toujours possible de supprimer les croisements d'un chemin pour obtenir un chemin plus court. Soit une ville d'indice i dans le chemin C, on note $s(i)$ la ville suivante du parcours. En général $s(i) = i + 1$ sauf pour la dernière ville du chemin pour laquelle $s(i) = 0$.

Soit un chemin C dans lequel la liaison $[i, s(i)]$ croise la liaison $[j, s(j)]$ avec $i < j$, on peut produire un chemin plus court supprimant le croisement. Pour cela on conserve le chemin C jusqu'à i inclus, puis on parcourt $[j, s(i)]$ (cette portion du circuit initial est parcourue à l'envers) et on finit par $[s(j), n]$, où n est la dernière ville du chemin. On représente cette transformation sur la figure 3.

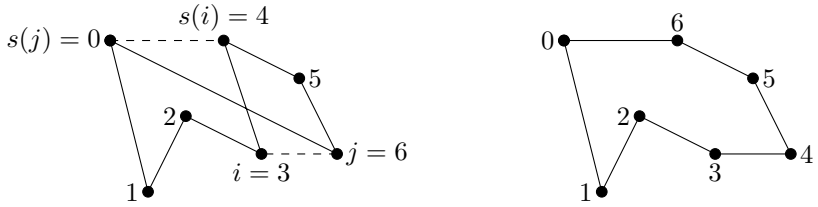

Fig. 3 : Transformation d'un chemin pour supprimer un croisement. On a supprimé les liaisons $[3, 4]$ et $[6, 0]$ et on a créé les liaisons en pointillés. La partie $[4, 5, 6]$ du chemin initial est inversée dans le chemin final

On pourrait rechercher dans le chemin C, tous les segments qui se croisent et les transformer de cette manière afin de faire disparaitre les croisements. Nous allons procéder de façon plus brutale en examinant toutes les paires (i, j) possibles pour déterminer quels changements réduisent la longueur du chemin. Cette méthode a l'avantage d'être plus simple à programmer sans augmenter considérablement le temps d'exécution. Cet algorithme d'optimisation s'appelle *2-opt*.

10. Écrire une fonction d'entête

```
permutation2opt(c:list, i:int, j:int) -> (list, float)
```

qui effectue la permutation des segments i et j décrite ci-dessus. Et qui renvoie le nouveau circuit ainsi que sa longueur.

11. Écrire une fonction d'entête

```
optimise2opt(c:list) -> (list, float)
```

qui prend en paramètre un circuit `c` et qui effectue les permutations de segments décrites ci-dessus jusqu'à ce qu'il n'en existe plus qui fasse diminuer la longueur du circuit. On atteint un nouveau minimum local pour la permutation de segments. La fonction renverra le nouveau circuit ainsi que sa longueur.

6.3 La touche finale

Arrivé ici, vous devriez déjà avoir des chemins tout à fait acceptables, nous allons introduire une dernière optimisation qui permet de gagner quelques kilomètres. Les chemins renvoyés par votre programme peuvent ressembler à celui de gauche dans la figure 4. Dans ce cas, il est clair que la ville numéro 6 doit être intégrée au segment $[1, 2]$ pour réduire la longueur totale du chemin.

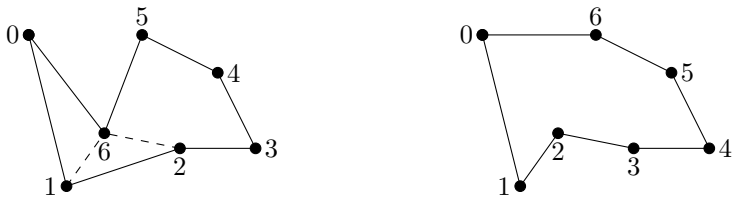

FIG. 4 : Dernière optimisation, il est clair que dans le chemin de gauche, la ville visitée en sixième position gagnerait à être insérée entre les villes 1 et 2.

Comme pour l'optimisation 2-opt de la partie précédente, nous allons procéder de manière brutale en testant l'intégration de tous les points du parcours à tous les autres segments en ne gardant que les modifications dont l'effet est de réduire la longueur du chemin.

12. Écrire une fonction d'entête

```
integrePointSegment(c:list, i:int, j:int) -> None
```

Qui modifie le chemin `c` en intégrant le point d'indice i au segment $(j, s(j))$.

13. Écrire une fonction d'entête

```
optimisation2(c:list) -> (list, float)
```

Qui effectue la transformation décrite ci-dessus jusqu'à ce qu'on ne puisse plus réduire la longueur du chemin c de cette manière. La fonction renvoie le nouveau chemin et sa longueur.

Voilà, il ne vous reste plus qu'à utiliser tous ces outils pour déterminer un chemin de distance minimale entre les villes de France. Le tableau ci-dessous vous donnera une idée de la qualité du chemin trouvé.

Longueur (km)	Commentaire
20 000	C'est nul, un enfant de 4 ans fait mieux que ça
10 000	Encore trop mauvais pour le bilan carbone du voyageur de commerce
7000	Ça commence à ressembler à quelque-chose, vous êtes sur la bonne voie
6500	Bien, on se rapproche d'une vrai solution
6000	Très bien !
5800	Excellent !
5788	Record personnel de Mr Schleck, si vous faites mieux il risque de le prendre mal…