TP1

Le TP sera écrit en Python. Pour l'affichage de la carte et du chemin proposé, on utilisera la librairie matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
import random
```

NB :

- permutation d'une liste d'entiers :

```
np.random.permutation(range(n))
```

- copie de liste :

```
copie_de_l = list(l)
```

→ important pour déterminer la liste des voisins

Problème du voyageur de commerce

Un voyageur de commerce doit visiter n villes au cours de sa tournée numérotées $0,...,n-1$. Ces villes sont définies par leurs coordonnées géographiques (x_i,y_i). Un parcours consiste à affecter un ordre à chaque ville :

- ordre $i \rightarrow s(i) ∈ 0..n-1$

sous la contrainte que chaque ville doit être visitée une fois et une seule. (autrement dit, s est une permutation de $(0,...,n-1)$ ⇒ $n!$ solutions). Le coût d'une solution s est : $$J(s) = \sum_{i=0}^{n-2} d(s(i),s(i+1)) + d(s(n-1), s(0))$$ avec: $$d(s(i), s(j)) = \sqrt{(x_{s(i)} - x_{s(j)})^2 + (y_{s(i)} - y_{s(j)})^2}$$

Le problème d'optimisation consiste à *minimiser* la distance totale.

Résolution par optimisation stochastique

Le but est d'écrire une librairie Python permettant de résoudre le problème du voyageur de commerce.

Un problème de voyageur de commerce sera décrit sous la forme d'une liste de villes avec leurs coordonnées géographiques.

Pour simplifier, on peut tirer les coordonnées de toutes les villes au hasard entre 0 et 1

pour les abcisses et les ordonnées

Une solution s sera définie come une permutation de la liste des n premiers entiers

Ecrire une fonction qui affiche graphiquement la solution courante (comme une série de segments reliant les points dans l'ordre parcouru).

Implémentez ensuite plusieurs méthodes de résolution. Vous devez impérativement implémenter

- la Méthode de Monte Carlo
- et la Méthode glouton.

Pensez en particulier écrire les fonctions suivantes:

- `randomVoisin(s)` : fonction qui retourne un voisin de la solution s pris au hasard
- `tousLesVoisins(s)` : fonction qui retourne la liste de tous les voisins de s
- `argmin_J(s)` : fonction qui retourne le voisin de s qui minimise la fonction J.

- Le voisin d'une solution s est obtenu en permutant deux éléments de s
- L'ensemble des voisins de s est l'ensemble des permutations de deux éléments possibles à partir de s.

Puis vous implementerez au choix :

- méthode tabou (hyperparamètre K)
- la méthode du recuit simulé (hypermaramètres β_0 et ε)

Tableau comparatif

Une fois vos méthodes testées, vous devez indiquer pour chaque méthode la longueur du chemin trouvé pour n = 10, n = 20, n = 40, n = 80 et n = 160, ainsi que le temps de calcul nécessaire pour obtenir la solution.

Vous afficherez graphiquement les différents chemins trouvés.

Rappels

Monte Carlo :

```
données : N
J* ← + infini
pour i de 1 à N:
    tirer un parcours s au hasard
    si J(s) < J* alors:
        J* ← J
```

```
        s* ← s
```

Glouton :

```
données : s
J* ← +infini
tant que J(s) < J*:
    J* <-- J(s)
    s <-- argmin_J(s)
```

Glouton aléatoire :

```
données : s
J* <-- +infini
Pour i de 1 à N:
    choisir au hasard s’ voisin de s
    si J(s’) < J(s) alors:
        J* <-- J
        s <-- s’
```

Tabou :

```
données : s,K
J* ← +infini
tabou ← []
Pour i de 1 à N:
     si J(s) < J* :
          J* ← J(s)
          s* ← s
     ajouter s à la fin de tabou
     supprimer le premier élément de tabou si |tabou| > K
     s ← argmin_J(s,tabou)
```

Recuit :

```
données : s, beta_0, eps
beta <--- beta_0
Pour i de 1 à N:
    choisir au hasard s’ voisin de s
    si J(s’) < J(s) alors:
        ACCEPTER
    sinon :
        p = exp(-beta * (J(s’)-J(s)))
        x <-- tirage_uniforme sur [0,1]
        si x < p alors:
            ACCEPTER
        sinon:
            REFUSER
    si ACCEPTER:
        s <-- s’
```

```
    beta <-- (1+eps) * beta
```