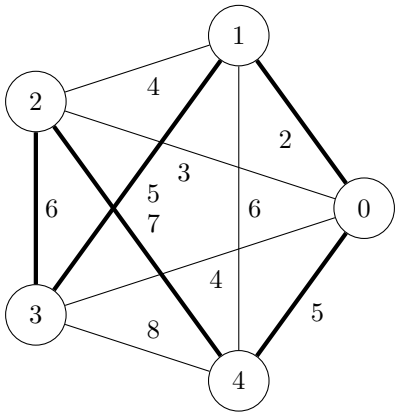
Algorithme probabiliste – voyageur de commerce et colonies de fourmis

Présentation

Le problème du *voyageur de commerce* (*Salesman problem*) s'énonce comme suit. On se donne un graphe non orienté pondéré (à pondérations strictement positives) complet (tous les sommets sont reliés entre eux) à n sommets. Il s'agit de trouver un cycle de poids total minimal passant par tous les sommets. C'est le problème que rencontre un voyageur de commerce qui doit visiter n villes en cherchant à minimiser la distance totale parcourue.



Un exemple de graphe complet à 5 sommets et un cycle passant par tous les sommets de poids total $2 + 5 + 6 + 7 + 5 = 27$.

Le problème est difficile (NP–complet) et l'algorithme naïf consistant à essayer tous les circuits possibles n'est pas envisageable quand le nombre de sommets augmente : il faut essayer $\frac{(n-1)!}{2}$ circuits (le sommet de départ n'import pas et le sens de parcours non plus).

Il existe plusieurs algorithme cherchant une approximation de la solution (un cycle de poids proche du minimum). Nous allons en proposer un qui s'inspire du comportement des colonies de fourmis recherchant un chemin entre leur habitat et une source de nourriture : on a observé qu'une colonie de fourmis ayant le choix entre deux trajets pour chercher de la nourriture avait tendance à utiliser le chemin le plus court. Un modèle a été proposé pour expliquer ce phénomène :

- Quand une fourmi trouve de la nourriture elle revient dans la colonie en laissant sur son chemin une piste de *phéromones*.
- Les autres fourmis passant à proximité de la piste sont attirées par cette piste et donc vont avoir tendance à la suivre de manière plus ou moins directe.
- En revenant à la colonie, ces fourmies vont elles aussi laisser des phéromones, *renforçant* ainsi la piste.

Si deux pistes sont possibles pour atteindre la même source de nourriture, la plus courte sera, dans le même temps, parcourue un plus grand nombre de fois, la piste de phéromones sera donc plus forte. Les phéromones étant volatiles, la piste la plus longue finira par s'effacer, de même, si la source de nourriture vient à se tarir, le chemin sera moins emprunté et la piste disparaîtra.

Algorithme des colonies de fourmis pour le voyageur de commerce

On s'inspire du comportement des fourmis pour concevoir un algorithme de résolution du problème du voyageur de commerce. Le principe est le suivant. Initialement, les arêtes du graphe sont enduites de la même quantité de phéromones.

- On envoie une fourmi à partir d'un sommet arbitraire.
- **Tant qu'**elle n'a pas fini son tour :
  elle choisit la prochaine ville parmi les villes non encore visitée suivant une certaine loi de probabilité. Une ville a d'autant plus de chance d'être choisie que le poids (distance) de l'arête qu'il faut emprunter est petite et qu'elle contient beaucoup de phéromones.

- Une fois le tour achevé, la fourmi dépose sur les arêtes du chemin parcourue une quantité de phéromone plus ou moins importante suivant la longueur du trajet.

Et on itère un certains nombre de fois l'opération, les pistes de phéromones s'évaporant légèrement à chaque itération.

Précisons un petit peu l'algorithme. Commençons par donner la loi de probabilité choisie pour effectuer le tour d'une fourmi. Les villes seront numérotées 0, 1, …, $n-1$ et on note respectivement $\Phi = (\varphi_{i,j})_{0\leqslant i,j\leqslant n-1}$ et $D = (d_{i,j})_{0\leqslant i,j\leqslant n-1}$ les matrices donnant la quantité de phéromones et le poids entre les villes

$$\varphi_{i,j} = \text{ phéromones entre } i \text{ et } j \qquad d_{i,j} = \text{ poids de l'arête } \{i,j\}$$

Choix de la prochaine ville. Imaginons que la fourmi se trouve dans une ville v, on note F l'ensemble des villes déjà visitées. Alors si $x \notin F$, la fourmi va choisir la ville x comme prochaine ville avec la probabilité p_x donnée par

$$p_x = \frac{\gamma + \varphi_{v,x}^{\alpha}\eta_{v,x}^{\beta}}{Total} \quad \text{avec } Total = \sum_{\substack{s=0\\ s\notin F}}^{n-1} \left(\gamma + \varphi_{v,s}^{\alpha}\eta_{v,s}^{\beta}\right) \quad \text{et où pour tout } i,j \text{ on a posé} \quad \eta_{i,j} = \frac{1}{d_{i,j}}$$

α et β sont ici des paramètres de l'algorithme permettant de donner plus ou moins d'importance aux phéromones ou au distances et γ est un paramètre encourageant l'exploration de nouvelles pistes.

Phéromones et évaporation. Si la fourmi a parcouru un tour $c = (x_0, x_1, \ldots, x_{n-1}, x_0)$ elle va déposer des phéromones tout au long du chemin. La quantité de phéromone sera la même tout au long du chemin et est donné par $\Delta\varphi$ défini par

$$\Delta\varphi = \frac{Q}{\pi(c)} \quad \text{où } \pi(c) \text{ est le poids total du chemin : } \pi(c) = d_{x_{n-1},x_0} + \sum_{i=0}^{n-2} d_{x_i,x_{i+1}}$$

Encore une fois, Q est ici un paramètre de l'algorithme. Enfin le dépot de phéromone se fait en modifiant la matrice Φ de la manière suivante :

$$\varphi_{i,j} \leftarrow \begin{cases} (1-\rho)\varphi_{i,j} & \text{si l'arête } \{i,j\} \text{ ne fait pas partie du chemin } c \text{ parcouru par la fourmi} \\ (1-\rho)\varphi_{i,j} + \Delta\varphi & \text{si l'arête a été parcourue par la fourmi} \end{cases}$$

ici $\rho \in [0,1[$ est un paramètre permettant de régler l'évaporation des phéromones.

L'algorithme fait ainsi intervenir un grand nombre de paramètres que l'on a regroupé dans le tableau suivant :

paramètre	**valeur**	
α	2	importance des phéromones dans le choix de la prochaine ville
β	1	importance de la distance dans le choix de la prochaine ville
γ	0.5	incitation à découvrir de nouvelles pistes
Q	1	quantité de phéromones déposées
ρ	0.1	évaporation des phéromones

Les valeurs données ne le sont qu'à titre indicatif, on pourra expérimenter avec d'autres valeurs des paramètres.

Implémentation

Implémentons l'algorithme, tout d'abord tous les paramètres de l'algorithme seront des variables globales. Ensuite, comme le suggère la présentation faites ci–dessus, il est naturel d'implementer le graphe sous la forme d'une matrice d'adjacence. Nous importerons ainsi le module `numpy` pour faire facilement des calculs matriciels. Il s'agit de programmer l'algorithme présenté plus haut. On peut ici procéder par étapes.

Trajet d'une fourmi. Pour calculer le tour que va effectuer la fourmi, il faut être en mesure de tirer une ville au hasard parmi les villes non encore visitées et en suivant une certaine loi de probabilité. Pour cela, on pourra utiliser la fonction `choice` du module `random` de `numpy`.

```
import numpy.random as rd

L = [1,2,3,4,5,6]  # liste des element a tirer aleatoirement
proba = [0.1, 0.15, 0, 0.15, 0.3, 0.3]  # liste des probas correspondantes
rd.choice(L,proba)  # renvoie un element de L aleatoirement suivant la loi de  proba
```

Ainsi pour simuler un cycle aléatoire d'une fourmi, on pourra :

- écrire une fonction `prochaineVille(villeCourante, nonVues, Phi, D)` qui étant donnée la ville courante `villeCourante`, la liste `nonVues` des villes non visitées et les matrices `Phi` et `D` renvoie la prochaine ville où va aller la fourmi.
- on peut alors facilement en déduire une fonction `tour(Phi,D)` qui procèdera de la manière suivante :
  - on choisit une ville de départ au hasard
  - on initialise une liste des villes non vue contenant toutes les villes sauf celle de départ
  - on initialise une liste `chemin` ne contenant que la ville de départ
  - **Tant qu'**on n'a pas fini le tour :
    - on choisit une ville parmi celle non visitées suivant les modalités vues précédemment
    - on rajoute la ville à la liste chemin et on la supprime de la liste des villes restantes.

    On oubliera pas de « *fermer* » le chemin ensuite.

Gestion des phéromones. Il s'agit ici de gérer l'évolution de la matrice Φ, plutôt que de faire les calculs coefficients par coefficients, on se propose de les faire matriciellement. Pour cela on a besoin des fonctions suivantes :

- `poids(cycle, D)` qui étant donné un cycle donné sous la forme d'une liste de sommets, renvoie le poids ce celui–ci.
- `matriceDeChemin(cycle,D)` qui étant donné un cycle `cycle` renvoie la matrice $M = (m_{i,j})_{0\leqslant i,j\leqslant n-1}$ définie comme suit :

$$m_{i,j} = \begin{cases} \frac{Q}{\pi(\texttt{cycle})} & \text{si l'arête } \{i,j\} \text{ est parcourue} \\ 0 & \text{sinon.} \end{cases}$$

Il est alors très simple de mettre à jour la matrice de phéromone en faisant des opérations matricielles.

Programmation finale et test

On peut dès lors finir d'implemnter l'algorithme et programmer une fonction `algoFourmis(D, nbIterations)` qui étant donné une matrice des distances (ou poids) `D` et un entier `nbIterations` donnant le nombre d'itération souhaité renvoie la liste des différents poids des parcours des fourmis au cours des itérations. On pourra ainsi visualiser la progression de ces poids en fil des itérations.

Pour tester votre fonction, vous pourrez utiliser les données des différentes villes de Djibouti (récupérées à l'adresse http ://www.math.uwaterloo.ca/tsp/data/index.html).

```
import numpy as np

D = np.load("djibouti.npy")
```

le fichier `djibouti.npy` doit se trouver dans le même répertoire que le fichier contenant votre code source. La matrice `D` a été préalablement calculée, on peut donc directement l'utiliser. La distance minimale pour ce problème est de 6656, voyez si l'algorithme se montre efficace.

Bonus. On peut améliorer l'algorithme de plusieurs façons :

- On peut « *lisser* » les résultats : au lieu de mettre à jour les phéromones à chaque trajet des fourmis, on peut envoyer des paquets de k fourmis en même temps et ne modifier Φ que lorsque les k fourmis ont fini leur tour.
- Le paramètre γ peut être dynamique : on peut inciter plus les fourmis à explorer de nouvelles pistes au début de l'algorithme et moins à la fin. Ou alors, si un minimum local semble atteint, augmenter γ pendant plusieurs tour pour chercher d'autres pistes.
- On pourra aussi tester l'algorithme sur de plus gros problèmes issus du site de l'université de Waterloo http ://www.math.uwaterloo.ca/tsp/data/index.html. Attention, il faudra dans ce cas, récupérer les données et calculer la matrice `D` correspondante.