Université Clermont Auvergne
Recherche opérationnelle
DUT Info 2A

Florent Foucaud

2021-2022

TP3 : Algorithmes heuristiques pour le voyageur de commerce

Dans ce TP nous allons étudier en pratique le problème du *voyageur de commerce* (en anglais : *Travelling Salesperson Problem*, généralement abbrévié "TSP"). On a un ensemble de villes avec les distances qui les séparent. On veut trouver une tournée pour parcourir toutes les villes et revenir au point de départ, en minimisant la distance totale parcourue. Par exemple, avec les villes suivantes :

	Clermont-Ferrand	Bordeaux	Toulouse	Lyon	Marseille
Clermont-Ferrand	0	376	377	167	475
Bordeaux	376	0	244	556	646
Toulouse	377	244	0	538	404
Lyon	167	556	538	0	314
Marseille	475	646	404	314	0

On peut faire par exemple la tournée Clermont $\rightarrow$ Toulouse $\rightarrow$ Bordeaux $\rightarrow$ Marseille $\rightarrow$ Lyon $\rightarrow$ Clermont pour $377 + 244 + 646 + 314 + 167 = 1748$km.

On va coder en Python des algorithmes heuristiques pour calculer une tournée aussi courte que possible. En Python, la tournée pourra être par exemple une liste d'entiers (chacune des n villes étant représentée par un entier entre 0 et $n-1$). Les distances seront stockées dans une matrice (liste de listes) `M` où `M(i,j)` est la distance entre les villes n°i et n°j. (Voir en bas du document pour des exemples.)

L'utilisation de Sagemath est optionnelle mais recommandée : je vous propose en bas du document, une fonction écrite pour Sagemath qui affiche graphiquement une tournée.

Exercice 1 (Algorithme glouton)**.**

On peut générer une tournée de façon gloutonne, de la façon suivante :

1. Choisir une première ville i (par exemple $i = 0$, ou au hasard avec `randrange(0,n)` qui renvoie un entier entre 0 et $n-1$)
2. initialiser le tour : $T[0] = i$
3. Tant qu'il reste des villes non visitées (pas encore placées dans T) :
   - choisir la ville j non encore visitée qui minimise la distance à la dernière ville visitée
   - ajouter j à T
4. renvoyer T

Coder une fonction `tournee_gloutonne(M)` qui prend en paramètre la matrice des distances `M` et calcule une tournée avec l'algorithme

Exercice 2 (Descente de gradient)**.**

On va utiliser maintenant la *descente de gradient*, algorithme méta-heuristique vu en cours.

On définit d'abord une opération de transformation d'une solution (ici, une tournée) en une autre, proche. Pour une solution T, soit $N(T)$ l'ensemble des solutions "voisines". Le principe est le suivant :

1. On génère une première solution T_0
   $T \leftarrow T_0$ #solution courante
2. Tant que $N(T)$ contient une solution meilleure que T :
   - choisir une nouvelle solution T' dans $N(T)$
   - $T \leftarrow T'$
3. Retourner T

Pour le voyageur de commerce, l'opération de transformation peut être définie de la façon suivante : étant donnée une tournée T, on prend deux villes et on échange leur position dans T pour obtenir un nouveau tour T'.

Coder l'algorithme de descente de gradient `DescenteGradient(M)` qui prend une matrice M des distances en entrée et renvoie une tournée. Vous pourrez d'abord :

- Coder une fonction `eval(T,M)` qui prend une solution `T` et la matrice `M` et renvoie la longueur du tour qui correspond à `T`.
- Coder une fonction `voisinage(T)` qui prend une solution `T` et renvoie la liste des solutions voisines.

Pour la solution initiale T_0, on pourra prendre au choix :

- pour les paresseux ou les pressés, la tournée `[0,...,n-1,0]` où `n` est le nombre de villes.
- une tournée calculée avec l'algorithme glouton de la question précédente.
- une tournée aléatoire (pour cela coder une fonction `tournee_aleatoire(M)` qui renvoie une tournée aléatoire).

Testez l'algorithme sur les données présentées en fin du document, et essayez-le à partir de différentes tournées de départ.

Exercice 3 (Recuit simulé)**.**

La descente de gradient peut être affinée avec la méthode du *recuit simulé* vue en cours. Dans cette modification de la descente de gradient, on a une certaine probabilité de prendre une "mauvaise" solution. Cette probabilité baisse au fil du temps (ceci est modélisé par une valeur de "température" qui baisse à chaque itération).

1. On génère une première solution S_0
   $S \leftarrow S_0$ #solution courante
   $i \leftarrow 0$ #compteur du nombre d'itérations
   $T \leftarrow 100$ #température initiale
2. Tant que $i < 1000$:
   - $T \leftarrow 0.99T$ #la température baisse
   - $i \leftarrow i + 1$
   - choisir au hasard une nouvelle solution S' dans $N(S)$
   - si S' est meilleure que S OU $random(0,1) < exp(-(|S'| - |S|)/T)$ #$|S|$ désigne la longueur de S
     $S \leftarrow S'$
3. Retourner S

Dans cet algorithme on peut faire varier les paramètres pour obtenir des résultats différents (nombre d'itérations, fonction de probabilité, température initiale). Dans Sage, `randrange(0,10000)/10000` renverra un flottant au hasard entre 0 et 1 (parmi 10000 valeurs possibles).

Coder l'algorithme de descente de gradient avec recuit simulé `RecuitSimule(M)` qui prend une matrice M des distances en entrée et renvoie une tournée.

Exercice 4 (Algorithme génétique)**.**

On rappelle le principe général de cette technique :

- Population initiale : un ensemble de solutions (aléatoires)

- On réitère x fois :
  - On sélectionne les p solutions les meilleures de la population courante (cela représente par exemple les meilleurs 20%, ou 50%, au choix)
  - On "croise" des paires aléatoires de ces solutions pour obtenir la population suivante (aussi de taille p). Chaque paire pourra avoir un ou plusieurs "enfants".

  - Des "mutations" apparaissent aléatoirement parmi les nouvelles solutions

Dans le cas du problème du voyageur de commerce, le "croisement" de deux solutions peut être fait de la façon suivante. Pour deux "parents" $p = [p_1, \ldots, p_n]$ et $m = [m_1, \ldots, m_n]$, on choisit un nombre aléatoire i entre 1 et $n - 1$. Un enfant pourra être construit en prenant le début $p_1, \ldots, p_i$ de la tournée "père" puis, en considérant le reste des villes dans l'ordre de la tournée de la mère. Par exemple, avec les tournées $p = [1, 3, 4, 0, 2, 5]$ et $m = [0, 2, 5, 3, 1, 4]$, avec $i = 2$, on obtiendra l'enfant $[1, 3, 0, 2, 5, 4]$. On peut avec une méthode similaire obtenir les enfants $[3, 1, 4, 0, 2, 5]$ (où au contraire on a gardé la deuxième partie de la tournée paternelle et complété le début avec l'ordre de visite maternel), et $[0, 2, 1, 3, 4, 5]$ (où le rôle du père et de la mère sont inversés).

Une "mutation" peut être faite en échangeant deux villes prises au hasard (mais pas la ville de départ) dans la tournée.

On pourra fixer une probabilité de croisement entre deux des meilleures solutions, et une probabilité de mutation (en principe, assez faible) des nouvelles solutions obtenues.

Les paramètres x (nombre de générations) et p (taille de la population) devront être fixés en faisant des tests pour trouver de bonnes valeurs.

Coder l'algorithme génétique `AlgoGenetique(M)` qui prend une matrice M des distances en entrée et renvoie une tournée.

Données et affichage d'une tournée dans Sagemath

Exemple de données avec 5 villes :

```
distances_5villes=[[0,376,377,167,475],[376,0,244,556,646],
[377,244,0,538,404],[167,556,538,0,314],[475,646,404,314,0]]

noms_5villes={0:'Clermont', 1:'Bordeaux', 2:'Toulouse', 3:'Lyon', 4:'Marseille'}

coordonnees_5villes={0:(50,47), 1:(24,40), 2:(37,28), 3:(62,46), 4:(65,23)}
```

Exemple de données avec 19 villes :

```
distances_19villes=
[[0,182,830,918,822,841,974,393,719,705,884,520,863,738,895,627,781,1138,285],
[182,0,648,619,675,659,792,216,842,555,840,338,819,546,713,445,599,907,250],
[830,648,0,793,1051,536,719,610,866,1219,1379,310,1358,566,730,245,503,1002,855],
[918,619,793,0,279,510,466,402,197,523,597,586,876,300,276,562,425,304,679],
[822,675,1051,279,0,774,771,478,109,280,318,741,297,585,555,747,698,506,558],
[841,659,536,510,774,0,284,520,667,974,1092,398,1071,201,318,329,84,650,844],
[974,792,719,466,771,284,0,602,664,1005,1089,584,1068,215,192,506,216,498,926],
[393,216,610,402,478,520,602,0,366,609,797,300,776,369,514,372,444,690,324],
[719,842,866,197,109,667,664,366,0,326,425,598,404,462,473,640,591,434,506],
[705,555,1219,523,280,974,1005,609,326,0,206,909,185,772,799,958,898,769,407],
[884,840,1379,597,318,1092,1089,797,425,206,0,1069,21,915,873,1065,1015,824,586],
[520,338,310,586,741,398,584,300,598,909,1069,0,1048,379,549,106,357,795,540],
[863,819,1358,876,297,1071,1068,776,404,185,21,1048,0,894,852,1044,995,803,565],
[738,546,566,300,585,201,215,369,462,772,915,379,894,0,147,331,117,442,678],
[895,713,730,276,555,318,192,514,473,799,873,549,852,147,0,498,233,330,824],
[627,445,245,562,747,329,506,372,640,958,1065,106,1044,331,498,0,290,771,652],
[781,599,503,425,698,84,216,444,591,898,1015,357,995,117,233,290,0,566,768],
[1138,907,1002,304,506,650,498,690,434,769,824,795,803,442,330,771,566,0,940],
[285,250,855,679,558,844,926,324,506,407,586,540,565,678,824,652,768,940,0]]

noms_19villes={0:'Biarritz', 1:'Bordeaux', 2:'Brest', 3:'Dijon', 4:'Grenoble',
 5:'Le Havre', 6:'Lille', 7:'Limoges', 8:'Lyon', 9:'Marseille', 10:'Monaco',
11:'Nantes', 12:'Nice', 13:'Paris', 14:'Reims', 15:'Rennes', 16:'Rouen',
17:'Strasbourg', 18:'Toulouse'}

coordonnees_19villes={0:(16,28), 1:(24,40), 2:(2,78), 3:(64,62), 4:(68,40),
5:(33,85), 6:(53,94), 7:(37,49), 8:(62,46), 9:(65,23), 10:(80,27), 11:(20,64),
12:(78,25), 13:(47,77), 14:(59,81), 15:(21,73), 16:(39,84), 17:(82,75), 18:(37,28)}
```

La fonction suivante permettra d'afficher une tournée dans Sagemath, étant donnée la tournée, les coordonnées des villes sur une carte, et les noms des villes.

```
def afficher_tournee(T,coordonnees_villes,noms_villes):
    L=[]

    for i in T:
        L.append(coordonnees_villes[i])

    p=plot(line(L))

    for i in T:
        p += plot(text(noms_villes[i],coordonnees_villes[i],
                                        bounding_box={'boxstyle':'round', 'fc':'w'}))

    p.show(axes=False)

T=[0,3,1,2,4,0]
afficher_tournee(T,coordonnees_5villes,noms_5villes) #test
```