Dipayan Chakraborty
Florent Foucaud
Malika More
Adrien Wohrer

Université Clermont Auvergne
Méthodes d'optimisation
BUT Info 2A
2023-2024

TP3 : Algorithmes heuristiques pour le voyageur de commerce
Version avec Solution

Dans ce TP nous allons étudier en pratique le célèbre problème du *voyageur de commerce* (en anglais : *Travelling Salesman Problem*, généralement abrévié "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 → Toulouse → Bordeaux → Marseille → Lyon → Clermont pour $377 + 244 + 646 + 314 + 167 = 1748$km.

On va coder dans ce TP des algorithmes heuristiques pour calculer une tournée aussi courte que possible. La tournée sera représentée par 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.

L'utilisation de Python et Sagemath est optionnelle mais recommandée : il est proposé ci-dessous, une fonction écrite pour Sagemath qui affiche graphiquement une tournée. Vous pouvez utiliser un autre langage que Python si vous préférez, mais dans ce cas vous ne pourrez pas utiliser la fonction d'affichage proposée, et il n'est pas garanti que vos chargés de TP puissent vous aider au niveau du code.

Exemple de données avec 5 villes (voir en bas du document pour des jeux de données plus importants) :

```
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)}
```

La fonction suivante permettra d'afficher une tournée dans Sagemath, étant donnée la tournée, les coordonnées des villes en 2D, et les noms des villes.

```
def afficher_tournee(T,coordonnees_villes,noms_villes,taille_texte):
    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'},
                                  fontsize=taille_texte))
    p.show(axes=False)

T=[0,3,1,2,4,0]
afficher_tournee(T,coordonnees_5villes,noms_5villes,10) #test
```

Exercice 1 (Algorithme glouton par ville la plus proche)**.**

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

1. Choisir une première ville i (par exemple $i = 0$, ou bien, au hasard avec `randrange(0,n)` qui renvoie un entier entre 0 et $n - 1$)
2. initialiser la tournée : $T[0] = i$
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. ajouter la ville de départ i à la fin de T pour clôre la tournée
5. renvoyer T

Coder une fonction `tournee_gloutonne(M)` qui prend en paramètre la matrice des distances `M` et calcule, puis renvoie, une tournée avec l'algorithme. (Par exemple, on testera la fonction en lançant

```
T=tournee_gloutonne(distances_5villes)
afficher_tourne(T,coordonnees_5villes,noms_5villes,10)
```

Le nombre de villes pourra être obtenu avec `len(M)`. La distance entre la ville `i` et la ville `j` est obtenue par `M[i][j]` ou bien `M[j][i]`.

Solution.

```python
def tournee_gloutonne(M):
    n=len(M)
    T=[randrange(0,len(M))] #premiere ville au hasard
    while len(T)<n:
        meilleur_score=1000000000
        meilleure_ville=0
        derniere_ville=T[-1]
        for i in range(n):
            if i not in T:
                score = M[i][derniere_ville]
                if score < meilleur_score:
                    meilleur_score=score
                    meilleure_ville=i
        T.append(meilleure_ville)
        T.append(T[0]) #revenir au debut
    return T

T20a=tournee_gloutonne(distances_20villes)
print(eval(T20a,distances_20villes), T20a)
afficher_tournee(T20a,coordonnees_20villes,noms_20villes,8)
```

Listing 1 – Algo glouton

Pour les 20 villes, on obtient entre 5295 et 6353 en fonction du départ.
Pour les 48 villes, on obtient entre 37928 et 43778 en fonction du départ (optimum = 33523).

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", c'est-à-dire, qui peuvent être obtenues via une seule opération de transformation. Le principe est le suivant :

1. On génère une première solution T_0
   $T \leftarrow T_0$ #T : 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 leurs positions dans T pour obtenir une nouvelle tournée 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 de distances `M` et renvoie la longueur de la tournée qui correspond à `T`.
- Coder une fonction `echange(T,i,j)` qui prend une solution `T` et deux indices de villes `i` et `j`, et renvoie la tournée semblable à `T` mais où les villes en position `i` et `j` sont inversées.
- Coder une fonction `voisinage(T)` qui prend une solution `T` et renvoie la liste des solutions "voisines", en utilisant `echange(T,i,j)`.

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

- 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).
- pour les paresseux ou les pressés, la tournée `[0,...,n-1,0]` où `n` est le nombre de villes.

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

Pour bien pouvoir le tester, on conseille d'afficher, à chaque itération de l'algorithme, le numéro de l'itération et la valeur de la meilleure solution courante.

Solution.

```python
def eval(T,M):
    cout = 0
    for i in range(len(T)-1):
        cout += M[T[i]][T[i+1]]
    return cout

def echange(T,i,j):
    nouv_tournee=[]
    for k in range(len(T)-1):
        if k==i:
            nouv_tournee.append(T[j])
        elif k==j:
            nouv_tournee.append(T[i])
        else:
            nouv_tournee.append(T[k])
    nouv_tournee.append(nouv_tournee[0]) #derniere ville=premiere ville
```

```
    return nouv_tournee


def voisinage(T):
    R = []
    for i in range(len(T)-1):
        for j in range(i+1,len(T)-1):
            R.append(echange(T,i,j))
    return R

def tournee_aleatoire(M):
    T=[]
    while len(T)<len(M):
        next = randrange(0,len(M))
        if next not in T:
            T.append(next)
    T.append(T[0])
    return T

def descente_gradient(M):
    T = tournee_aleatoire(M)
    while(True):
        N = voisinage(T)
        modif = False
        for tournee in N:
            if eval(tournee,M) < eval(T,M):
                T = tournee
                modif = True
        if not modif:
            break
    return T

T20b=descente_gradient(distances_20villes)
print(eval(T20b,distances_20villes), T20b)
afficher_tournee(T20b,coordonnees_20villes,noms_20villes,8)
```

Listing 2 – Descente de gradient

Pour les 20 villes, on obtient entre 4190 et 6597 (sur 500 exécutions).
Pour les 48 villes, on obtient entre 42101 et 57495 (sur 50 exécutions). (optimum = 33523)