

T.P. d'informatique n° 7-3

Probleme du voyageur de commerce

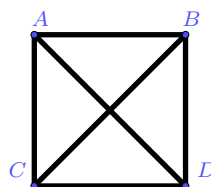
Ce TP (extrait du livre Informatique chez Dunod) est consacré à l'étude du parcours de graphes non orientés et à la résolution du problème du voyageur de commerce, tant de manière exacte par force brute dans des cas simples que de manière approchée à l'aide d'algorithmes génétiques dans des cas plus complexes.

Un graphe $\mathcal{G}$ est un **graphe hamiltonien** s'il possède au moins un cycle passant par tous les sommets de $\mathcal{G}$ exactement une fois ; un tel cycle est appelé **cycle hamiltonien**.

Un représentant de commerce part de sa ville d'origine et doit passer visiter ses clients dans des villes différentes, une fois et une seule. Il a bien sûr intérêt à minimiser la longueur du cycle qu'il va faire et il souhaite rentrer dans sa ville d'origine. Ce problème est appelé « problème du voyageur de commerce » (*salesman problem*) et est apparu dans les années 1930. Avec le vocabulaire introduit plus haut, il consiste à trouver, dans un graphe hamiltonien (pondéré), un cycle hamiltonien de longueur minimale.

Nous nous restreindrons aux graphes complets pour la suite du TP, c'est-à-dire un graphe non orienté tel que toute paire de sommets distincts est reliée, comme par exemple le graphe ci-contre.

Cela revient dans l'analogie du représentant de commerce à dire qu'il est possible d'aller de n'importe quelle ville à n'importe quelle autre sans faire nécessairement étape dans une ville intermédiaire d'éjà visitée. Par ailleurs, le graphe est non orienté, chaque chemin entre deux villes pouvant être emprunté indifféremment dans les deux sens. Un graphe complet est bien sûr toujours hamiltonien.



Le problème du voyageur de commerce est connu pour être particulièrement difficile : il est dans la classe de complexité des problèmes NP-complets, c'est-à-dire pour lequel on ne connaît pas d'algorithme répondant exactement au problème avec une complexité en temps polynomial.

Résolution par force brute

Une approche naïve (dite par force brute) de résolution du problème du voyageur de commerce consiste à tester toutes les boucles hamiltoniennes d'origine donnée, puis à sélectionner celle qui a la plus petite longueur.

- 1) Quelle est la complexité de cet algorithme en fonction du nombre n de sommets du graphe ?

A titre d'illustration, si on estime qu'il faut une microseconde pour tester un trajet, cela nous donne les durées d'exécution suivantes pour le nombre de villes indiqué

Nb de villes	Nb de possibilités	Temps de calcul
5	12	12 microsecondes
10	181 440	0,18 seconde
15	43 milliards	12 heures
20	60 E+15	1928 ans
25	310 E+21	9,8 milliards d'années (!)

- 2) Écrire une fonction récursive `genere_permutations(L)` prenant en argument une liste L et renvoyant la liste de toutes les permutations de ses éléments, chaque permutation étant elle-même codée dans une liste. Par exemple, `genere_permutations([0,1,2])` renverra (dans cet ordre ou un autre) : `[0,1,2]`, `[0,2,1]`, `[1,0,2]`, `[1,2,0]`, `[2,0,1]`, `[2,1,0]`

- 3) Écrire une fonction `genere_boucle(n , depart)` prenant en argument un entier `n` et un entier `depart` et renvoyant une liste de toutes les boucles hamiltoniennes possibles d'origine (et d'arrivée) `depart` sur un graphe complet d'ordre n dont les sommets sont numérotés par des nombres de 0 à $n - 1$. On se servira de la fonction précédente.
- 4) Écrire un fonction `distances_boucles(LB, T)` prenant en argument une liste de boucles hamiltoniennes `LB` codées par des listes d'entiers et la matrice des poids `T` du graphe complet où les boucles sont situées, et qui renvoie la liste des distances associées à chacune des boucles dans le même ordre que celui des boucles.
- 5) Écrire une fonction `meilleure_boucle(LB,T)` de mêmes arguments que `distances_boucles` renvoyant un tuple contenant l'indice de la boucle la plus courte et son poids total.
- 6) On dispose d'une liste de coordonnées de points repérés dans un repère orthonormé du plan affine euclidien, codée sous forme de tuples de longueur 2.
Écrire une fonction `coord_vers_marice(LC)` qui prend en argument une telle liste et renvoie une matrice des distances « à vol d'oiseau » entre les couples de points.
- 7) On dispose des coordonnées des points suivants :
 $A(0, 0)$, $B(1, 1)$, $C(2, 4)$, $D(1, -3)$, $E(0, -5)$, $F(0, 4)$, $G(-1, -5)$, $H(-2, 3)$ et $I(-3, 0)$.
Résoudre le problème du voyageur de commerce sur le graphe ayant ces points comme sommets, la distance étant la distance à vol d'oiseau. Le voyageur part du point A . Représenter les sommets et la solution obtenue.

Résolution par algorithme génétique

Le problème du voyageur de commerce ne peut être résolu par force brute : pour 25 villes disposées aléatoirement, il faut attendre près de 2 millénaires pour avoir la solution minimale avec cette méthode.

On cherche donc des solutions approchées, accessibles en un temps raisonnable.

La recherche est encore active en ce domaines et on peut trouver sur Internet plusieurs sites mettant proposant des défis aux participants consistant à trouver une solution optimale (dans le sens « meilleure que celles proposées par les autres ») au problème du voyageur de commerce (par exemple le défi des 250 villes sur <http://labo.algo.free.fr/index.html>).

Nous allons chercher une réponse approchée par un algorithme génétique : pour commencer nous allons créer une matrice de poids associée à un graphe.

Création de la matrice des poids

Nous allons travailler avec le fichier '`listeprefectureFrance.txt`' disponible sur le site cahier de prepa et contenant toutes les préfectures de France métropolitaine (privé des préfectures des départements d'Ile de France...)

- 8) On fournit le fichier '`listeprefectureFrance.txt`' contenant 87 lignes de 3 données séparées par un point virgule ';' : le nom de la ville puis les coordonnées GPS de cette ville, les préfectures de région se trouvant à la fin. A noter que si on veut représenter de façon classique ces villes sur un graphe, la première coordonnée GPS correspond à la latitude et donc l'ordonnée et la seconde à la longitude donc l'abscisse...
Écrire une fonction sans argument qui renvoie une liste des 87 tuples (ou listes) de la forme (x, y) correspondant aux coordonnées des villes.
- 9) Créer la matrice des poids (ici des distances à vol d'oiseau) associée à ce problème.

Algorithme génétique

Une des méthodes pour donner une solution approchée au problème du voyageur de commerce s'appuie sur des **algorithmes génétiques**.

On part d'une génération initiale d'individus aléatoires, **un individu étant ici une boucle hamiltonienne sur le graphe** et correspond donc à une solution approchée (pas forcément performante et encore moins optimale) du problème du voyageur de commerce.

On écrit ensuite une fonction permettant de créer la génération suivante à partir des principes suivants :

- ☞ plus un individu est performant (i.e plus la route qu'il emprunte est courte), plus il a d chances de se reproduire. Un tirage au sort entre individus reproducteurs est conduit selon le principe de la roulette, principe qui sera détaillé plus loin ;
- ☞ à chaque nouvelle génération, les individus peuvent voir leur code génétique muter avec une probabilité p , qui est un paramètre (inconnu...). Une mutation consiste à sélectionner deux indices dans le code génétique d'un individu et à inverser le code génétique de cet individu entre ces deux indices. Ici, cela signifie inverser le sens

du trajet entre deux villes dans la boucle hamiltonienne définissant l'individu (ce qui préserve évidemment le caractère hamiltonien du parcours) ;

Par exemple si on a l'individu $[5, 3, 2, 1, 4, 0, 8, 7, 6]$ et que l'on procède à une mutation entre les termes d'indice 1 et 5, on arrive à l'individu $[5, \mathbf{0}, \mathbf{4}, \mathbf{1}, \mathbf{2}, \mathbf{3}, 8, 7, 6]$.

- ☞ deux parents créent deux enfants par un système dit de *cross-over* : le premier donne le début le début (dont l'indice de fin est déterminé aléatoirement) de son parcours à un fils et le second parent donne le début de son parcours à l'autre fils. Les parcours des fils sont alors complétés par les parcours du parent dont le « code génétique » (ici, la boucle hamiltonienne qui le définit) n'a pas été utilisé, en ajoutant à la fin du code génétique du fils les sommets qui n'y apparaissent pas encore, pris dans l'ordre d'apparition dans le code du second parent.

Par exemple si on a tiré l'indice 4 et que les deux parents sont les individus

$p1 = [5, 3, 2, 1, 4, 7, 8, 0, 6]$ et $p2 = [3, 1, 0, 5, 8, 6, 4, 2, 7]$ alors les deux fils seront les individus

$f1 = [\mathbf{5}, \mathbf{3}, \mathbf{2}, \mathbf{1}, 0, 8, 6, 4, 7]$ et $f2 = [\mathbf{3}, \mathbf{1}, \mathbf{0}, \mathbf{5}, 2, 4, 7, 8, 6]$

- 10) La fonction `random.shuffle(L)` mélange aléatoirement la liste L (et retourne None).
En utilisant la fonction `random.shuffle`, écrire une fonction `cree_initiale(N)` qui renvoie la génération initiale, codée comme liste de listes. Cette génération comportera N individus, qui seront chacun représentés par une liste (des entiers de 0 à 86) contenant les indices des 87 villes dans un ordre aléatoire.
- 11) Écrire une fonction `meilleur_individu(population)` qui prend en argument une liste de listes d'individus représentés par leur parcours (que l'on oubliera pas de fermer), et qui renvoie un tuple contenant la distance totale parcourue par le meilleur individu de `population` et son parcours.
- 12) Écrire une fonction `mutation(individu)` qui prend en argument un individu décrit par son cycle hamiltonien et codé sous forme d'une liste et qui renvoie l'individu une fois son code génétique muté, comme décrit en début de cette partie. Les indices de mutations sont aléatoires, générés par `random.randint`.
- 13) Écrire une fonction `crossover(p1,p2)` prenant en argument deux individus parents `p1` et `p2` et renvoyant leurs deux fils, obtenus par le procédé de *cross-over* décrit en début de cette partie.
- 14) La roulette est une des façons de sélectionner les individus reproducteurs. Un individu donné a une probabilité proportionnelle à $f(d)$ de se reproduire où f est une fonction donnée décroissante et d la distance de parcours de l'individu.

Plus précisément, on dispose au départ d'une génération, codée sous forme de liste de listes. On calcule la liste D des distances totales parcourues par chaque individu, puis la liste F des $f(d)$. À partir de cette liste F , on crée une liste R , représentant la fonction de répartition de la variable aléatoire X , où $\mathbb{P}(X = x_k)$ est

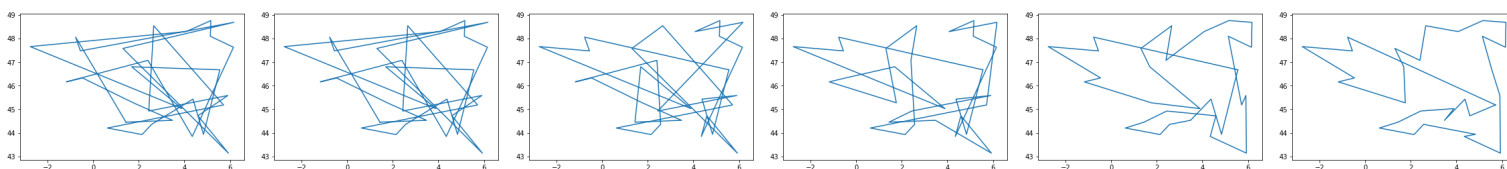
par définition la probabilité que l'individu d'indice k se reproduise. On a ainsi : $R_k = \frac{\sum_{j=0}^k F_j}{\sum_{j=0}^{N-1} F_j}$. On tire au

hasard un flottant x entre 0 et 1. L'entier k tel que $R_k \leq x < R_{k+1}$ donne l'indice de l'individu choisi pour se reproduire.

Écrire une fonction `genere_roulette(population, T)` qui prend en argument une génération et la matrice des poids T et qui renvoie la liste r décrite ici. On pourra prendre $f(d) = \frac{1}{d - 0.99m}^3$, m étant la distance parcourue par l'individu le plus performant de la génération considérée (il s'avère empiriquement que cette fonction donne des résultats convenables).

- 15) Écrire une fonction `indiceroulette(R)`, qui prend en argument la liste précédemment construite et qui renvoie l'indice d'un individu tiré au sort pour se reproduire.
- 16) Créer une fonction `generation_suivante` en utilisant les éléments précédents (génération par *cross-over* et mutation aléatoire).
- 17) La tester sur plusieurs générations en prenant soin de ne pas lancer de calculs trop longs (en travaillant par exemple sur une partie de la liste de ville et sur peu de générations).

Exemples de sorties sur 30 villes, des meilleurs individus des générations 1, 2, 4, 9, 40 et 100



Solution

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jan 27 19:59:06 2019

@author: FranÃ§ois
"""

import numpy as np

def genere_perm(L):
    n = len(L)
    if n==1:
        return [L]
    return [P[k:]+[L[n-1]] + P[:k] for k in range(n) for P in genere_perm(L[:n-1])]

def genere_boucle(n,depart):
    L= list(range(depart)) + list(range(depart+1,n))
    perm = genere_perm(L)
    return [[depart] + p + [depart] for p in perm]

def longueur_chaine(L,T):
    dist = 0
    for i in range(len(L)-1):
        a = T[L[i], L[i+1]]
        if a == -1 :
            return -1
        dist += a
    return dist

def distances_boucles(LB,T):
    return [longueur_chaine(B,T) for B in LB]

def meilleure_boucle(LB, T):
    poids = distances_boucles(LB, T)
    i, rec = 0, poids[0]
    for j in range(len(LB)):
        if poids[j] < rec :
            i, rec = j, poids[j]
    return i,rec

def coord_vers_matrice(LC):
    n = len(LC)
    M = np.zeros((n,n))
    for i in range(n-1):
        for j in range(i+1, n):
            M[i,j] = np.sqrt((LC[i][0] - LC[j][0])**2
                             + (LC[i][1] - LC[j][1])**2)
            M[j,i] = M[i,j]
    return(M)

LC = [[1.,0,], [1,1],[0,0]]

L = [(-0,0),(1,1), (2,4), (1,-3),
      (0,-5),(0,4),(-1,-5),(-2,3),(-3,0)]

M = coord_vers_matrice(L)
LB = genere_boucle(len(L),0)
indmin, lmin = meilleure_boucle(LB, M)
print(indmin, lmin)
```

```

64
65 import matplotlib.pyplot as plt
66 X = [L[i][0] for i in range(9)]
67 Y = [L[i][1] for i in range(9)]
68 plt.scatter(X,Y)
69 CB = [L[i] for i in LB[indmin]] #meilleur cycle hamiltonien
70 X1 = [CB[i][0] for i in range(10)]
71 Y1 = [CB[i][1] for i in range(10)]
72 plt.plot(X1, Y1, 'bo-')
73 plt.show()
74
75 f = open('villes250.txt','r')
76 villesqcq=[]
77
78 for ligne in f:
79     x,y = ligne.split(';')
80     villesqcq.append((float(x),float(y)))
81
82 f.close()
83
84 print(villesqcq[1:6])
85
86
87 X = [villesqcq[i][0] for i in range(100)]
88 Y = [villesqcq[i][1] for i in range(100)]
89 plt.scatter(X,Y)
90 plt.show()
91
92
93 f = open('listeprefectureFrance.txt','r')
94 villes=[]
95 Nomvilles=[]
96 for ligne in f:
97     M,x,y = ligne.split(';')
98     villes.append((float(x),float(y)))
99     Nomvilles.append(M)
100 f.close()
101
102 print(villes[1:6])
103
104
105 Y = [villes[i][0] for i in range(len(villes))]
106 X = [villes[i][1] for i in range(len(villes))]
107
108
109 plt.plot(X,Y,'bo')
110 plt.axis('equal')
111 for i in range(1,15):
112     plt.text(X[-i], Y[-i], Nomvilles[-i])
113 plt.show()
114
115
116
117 M = coord_vers_matrice(villes)
118
119 import random as rd
120
121 def cree_initiale(N,nbvilles=30):
122     Pop = []
123     for i in range(N):
124         a = list(range(0, nbvilles))
125         np.random.shuffle(a)
126         Pop.append(a)
127     return(Pop)
128

```

```

12 p= 0.5
130
131 def meilleur_individu(population,M):
132     i,p = meilleure_boucle(population, M)
133     return population[i],p # on oublie p
134
135 def mutation(individu, probamut = p):
136     x = rd.random()
137     if x < probamut : #on mute
138         newind = individu[:]
139         n = len(individu)
140         a,b = rd.sample(list(range(n)),2)
141         a,b = min(a,b), max(a,b)
142         T = newind[a:b]
143         T.reverse()
144         return newind[:a] + T + newind[b:]
145     return individu
146
147
148 def crossover(p1, p2):
149     indice = rd.randint(1, len(p1)-1)
150     fils1, fils2 = p1[:indice], p2[:indice]
151     for j in range(len(p1)):
152         if p2[j] not in fils1[:indice]:
153             fils1.append(p2[j])
154         if p1[j] not in fils2[:indice]:
155             fils2.append(p1[j])
156     return(fils1, fils2)
157
158
159 def genereroulette(population, M):
160     ch = min([longueur_chaine(i,M) for i in population])
161     S = [1/(longueur_chaine(i,M) - ch*0.99)**3 for i in population]
162     somme = sum(S)
163     S = [i/somme for i in S]
164     R = [S[0]]
165     for i in range(len(S)-1):
166         R.append(R[i]+S[i+1])
167     return R
168
169
170 def indiceroulette(R):
171     indice, a = 0, rd.random()
172     while indice < len(R):
173         if a<= R[indice]:
174             return indice
175         indice += 1
176     return indice - 1 # normalement on ne passe pas par l'Ã
177
178
179 def generation_suivante(population, probamut, M):
180     newgen = []
181     R = genereroulette(population, M)
182     for a in range(len(population)//2):
183         i,j = indiceroulette(R), indiceroulette(R)
184         fils1, fils2 = crossover(population[i], population[j])
185         newgen.extend([fils1, fils2])
186     for i in range(len(newgen)):
187         newgen[i] = mutation(newgen[i],probamut)
188     return newgen
189
190 def generationssuccessives(N,nbiter,probamut,M):
191     nvilles, Pop = len(M), cree_initiale(N)
192     L = [meilleur_individu(Pop, M)[0]]
193     for i in range(nbiter):

```

```

194     Pop = generation_suivante(Pop, probamut,M)
195     L.append(meilleur_individu(Pop, M)[0])
196     print(i, longueur_chaine(L[-1],M))
197     return L
198
199 nbgen = 1000
200 L = generationssuccessives(50,nbgen,0.06,M)
201
202 def dessinechemin(villes,L,k):
203     Y = [villes[i][0] for i in L[k]]
204     X = [villes[i][1] for i in L[k]]
205     Y.append(villes[L[k][0]][0])
206     X.append(villes[L[k][0]][1])
207     plt.plot(X,Y)
208     plt.show()
209
210 for k in range(int(np.sqrt(nbgen))):
211     dessinechemin(villes,L,k*k)

```