Devoir surveillé n°4

Durée : 45 minutes

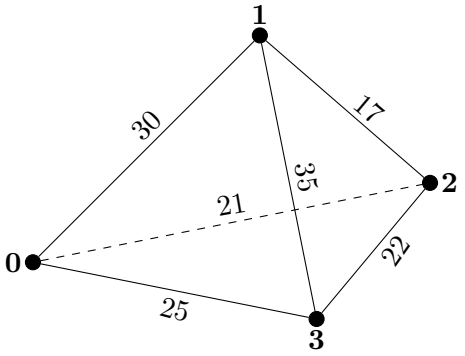
L'utilisation d'une calculatrice ou de tout document est interdite.
Toute sortie anticipée est interdite.

Introduction

Nous abordons le *problème du voyageur de commerce* ; c'est un problème important par son large champ d'application, mais dont on ne connaît aucune méthode de résolution exacte efficace. Nous mettons en place une résolution *approchée* grâce à une méthode générale appelée *algorithme génétique*. C'est une méthode de résolution approchée de problème difficile qui s'inspire de l'évolution darwinienne en constituant une population qui évolue par reproduction et sélection afin de satisfaire de mieux en mieux au critère de sélection.

1) Le problème du voyageur de commerce

Soit $N \in \mathbb{N}^*$; considérons un représentant de commerce situé initialement dans une ville (numérotée 0) et qui doit effectuer sa tournée en visitant successivement une et une seule fois N villes (numérotées $1, 2, \ldots, N$) avant de revenir à sa ville 0. Le problème consiste, connaissant la distance séparant chacune des $N+1$ villes des autres, à déterminer un ordre de visite pour lequel la longueur de la tournée (c'est-à-dire la distance parcourue) soit minimale. La tournée effectuée sera alors dite *minimale*.
Par exemple, supposons que $N = 3$. La situation peut se représenter à l'aide d'un graphe ayant un sommet par ville (numérotés 0, 1, 2, 3) et une arête reliant chaque ville i à chaque autre ville j et valuée (c'est-à-dire munie d'une valeur) par la distance $D_{i,j}$ séparant les deux villes. Le graphe peut se donner par une matrice carrée D ayant 4 lignes (numérotées de 0 à 3) et 4 colonnes (numérotées de 0 à 3) et dont l'élément ligne i, colonne j est la distance $D_{i,j}$ séparant la ville i de la ville j.



$$\begin{array}{c} \begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} 0 & 30 & 21 & 25 \\ 30 & 0 & 17 & 35 \\ 21 & 17 & 0 & 22 \\ 25 & 35 & 22 & 0 \end{pmatrix} \begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \end{array}$$

Une tournée est représentée par la suite des villes empruntées. Ici, les tournées possibles sont :

$$(0,1,2,3,0) \quad (0,1,3,2,0) \quad (0,2,1,3,0) \quad (0,2,3,1,0) \quad (0,3,1,2,0) \quad (0,3,2,1,0).$$

- pour la tournée : (0,1,2,3,0), la longueur est : $D_{0,1} + D_{1,2} + D_{2,3} + D_{3,0} = 30 + 17 + 22 + 25 = 94$
- pour la tournée : (0,3,2,1,0), la longueur est : $D_{0,3} + D_{3,2} + D_{2,1} + D_{1,0} = 25 + 22 + 17 + 30 = 94$

Les longueurs des deux tournées sont ici égales puisque la seconde tournée se déduit de la première en parcourant les villes en sens inverse.

Q1. Quelles sont les longueurs des deux tournées : (0,1,3,2,0) et (0,2,3,1,0) ?

On pourrait naïvement penser qu'une tournée minimale s'obtiendrait en suivant comme stratégie de se diriger toujours au départ d'une ville i vers la ville j la plus proche non encore visitée.

Q2. Quelle tournée produirait cette stratégie dans le cas ci-dessus ? Est-elle minimale ?

Une tournée est déterminée par l'ordre de visite des villes 1 à N ; il y en a exactement $N!$. Une stratégie naïve par force brute consistant à comparer toutes les $N!$ tournées possibles devient rapidement totalement impraticable pour des valeurs de l'entier N pas trop petites.

Par exemple sur un ordinateur puissant cadencé à 10Ghz ce calcul prendrait pour :

$N = 20$: plus de $\dfrac{20!}{10^{10} \times 3600 \times 24 \times 365} \approx 7$ ans.

$N = 30$: plus de $\dfrac{30!}{10^{10} \times 3600 \times 24 \times 365} \approx 800\,000$ milliards d'années.

2) Implémentation et représentation des données

a) Initialisation du problème du voyageur de commerce

On fixe la valeur de l'entier N, nombre de villes à visiter, par exemple $N = 20$ dans une variable `N` globale de type entier.

```
N = 20 # Nombre de villes à visiter
```

On se donne les distances qui séparent les $N+1$ villes (de 0 à N) dans une matrice carrée D ayant $(N+1)$ lignes et $(N+1)$ colonnes ; la matrice est définie par une variable globale `D`, liste de $(N+1)$ listes de $(N+1)$ éléments ; l'élément `D[i][j]` représente la distance $D_{i,j}$ séparant les villes i et j :

```
# Matrice des distances inter-villes
D = [[ 0, 24, 18, ... , 43, 42, 38],
     [24, 0, 13, ... , 24, 14, 7],
     ...
     [38, 7, 23, ... , 14, 10, 0]]
```

Q3. Quelle valeur prend `D[1][2]` ? Quelle distance sépare les villes 19 et 20 ?

Une tournée est représentée par la liste donnant l'ordre de visites des N villes, c'est-à-dire en omettant les 0 en début et en fin de tournée ; par exemple :

La tournée (0,1,2,3,...,19,20,0) est représentée par la liste [1,2,3,..,19,20].

La liste [20,19,...,3,2,1] représente la tournée (0,20,19,...,3,2,1,0).

On peut alors écrire le code d'une fonction prenant en argument une tournée (liste donnant l'ordre des visites des N villes) et qui renvoie la longueur de la tournée :

```
1 def longueur(tournee):
2     lgr = D[0][tournee[0]]
3     for k in range( ... ): # à compléter
4         lgr = lgr + D[tournee[ ... ]][tournee[ ... ]] # à compléter
5     lgr = lgr + D[tournee[N-1]][0]
6     return lgr
```

Q4. Compléter les lignes 3 et 4 du code de la fonction `longueur` ci-dessus.

b) Constitution d'une population

Soit N_p un entier assez grand. On génère aléatoirement une population constituée de N_p individus, chaque individu représentant une tournée.

Pour générer aléatoirement une tournée on applique la fonction `shuffle()` du module `random` (qu'on aura importée), qui, avec en argument une liste, change aléatoirement l'ordre des éléments de la liste.

```
>>> L = [1, 2, 3, ..., 20]
>>> shuffle(L)
>>> L
    [13, 1, 7, ..., 15]
```

La population générée sera stockée dans un dictionnaire `Population`, sous la forme de couples (clé, valeur) où valeur est une tournée et clé sera sa longueur.

La fonction `PopulationInitiale` renvoie avec en argument un entier `Np`, un tel dictionnaire contenant une population de N_p individus générés aléatoirement.

```
1 def PopulationInitiale(Np):
2     T0 = [ k for k in range(1,N+1) ]   # T0 = [1, 2, ..., N]
3     Population = {}                    # Création du dictionnaire
4     for k in range(Np):                # Insertion des individus dans le dictionnaire
5         T1 = T0[:]                     # Copie de T0
6         shuffle(T1)                    # Permutation de T1 : tournée aléatoire
7         cle = longueur(T1)             # La clé sera la longueur
8         Population[cle] = T1 # Insertion du nouvel individu de valeur T1
9     return Population
```

Cette représentation a pour défaut qu'on risque d'obtenir une population de moins de N_p individus puisque des tournées différentes peuvent avoir même longueur. Mais ce n'est pas problématique ici.

Q5.1. Pourquoi ne peut-on pas plutôt prendre pour clé la tournée et pour valeur sa longueur ?

Q5.2. Quel phénomène inattendu se produirait si on changeait à la ligne 5 : `T1 = T0[:]` par `T1 = T0` ?

3) Implémentation de l'algorithme génétique

Le principe de l'algorithme génétique est le suivant :

- On constitue une population initiale de N_p individus.
- *Sélection* : on sélectionne parmi eux les N_s individus de moindres longueurs (avec $N_s < N_p$).
- *Reproduction* : parmi cette sélection on tire au hasard N_p couples d'individus, qui chacun donneront naissance à deux enfants par croisement et mutation aléatoire ; on obtient ainsi une nouvelle génération de $2N_p$ descendants.
- *Nouvelle population* : on ajoute la population des $2N_p$ descendants à la population des N_s sélectionnés ; puis on reprend à l'étape de sélection avec cette nouvelle population.

On répète ce procédé un grand nombre de fois. Au fur et à mesure des générations, apparaissent des individus de longueur de plus en plus courte. À la fin du processus on sélectionne dans la population obtenue l'individu de longueur minimale. C'est la solution approchée recherchée.

a) Sélection

Afin de sélectionner les individus les mieux adaptés, on a besoin d'un algorithme de tri. On utilisera un tri par insertion.

Q6. Parmi les 4 fonctions proposées laquelle procède à un tri par insertion dans l'ordre croissant de la liste passée en argument ? On ne demande pas de justifier la réponse.

```
def triCroissant_1(L):
    n = len(L)
    for i in range(1,n):
        elt = L[i]
        k = i
        while k>0 and L[k-1] > elt:
            L[k] = L[k-1]
            k -= 1
        L[k] = elt
```

```
def triCroissant_2(L):
    n = len(L)
    for i in range(1,n):
        elt = L[i]
        k = i
        while k>0 and L[k-1] < elt:
            L[k] = L[k-1]
            k -= 1
        L[k] = elt
```

```
def triCroissant_3(L):
    n = len(L)
    for i in range(1,n):
        elt = L[i]
        k = i
        while k>0 and L[k-1] > elt:
            L[k-1] = L[k]
            k -= 1
        L[k] = elt
```

```
def triCroissant_4(L):
    n = len(L)
    for i in range(1,n):
        elt = L[i]
        k = i
        while k>0 and L[k-1] < elt:
            L[k-1] = L[k]
            k -= 1
        L[k] = elt
```

Dans la suite, cette fonction sera appelée `triCroissant()`.

Pour procéder à la sélection, on regroupe dans une liste toutes les clés (= longueurs) des individus de la population. On trie les éléments de cette liste dans l'ordre croissant, pour ne conserver que les N_s premiers. Puis on reconstitue le dictionnaire des individus dont les clés ont été conservées.

```
1 def selection(Population, Ns):
2     MoindresLongueurs = [ cle for cle in Population ] # Liste des clés
3     triCroissant(MoindresLongueurs) # Tri croissant de liste des clés
4     MoindresLongueurs = MoindresLongueurs[:Ns] # Sélection des Ns moindres
5     Dict = {} # Constitution du dictionnaire
6     for cle in ... : # à compléter
7         Dict[cle] = ... # à compléter
8     return Dict
```

Q7. Compléter les lignes 6 et 7 du code ci-dessus comme indiqué.

À la fin de l'algorithme, on aura aussi besoin d'une fonction prenant en argument le dictionnaire de la population obtenue et qui renvoie l'individu de clé minimale. On effectue pour cela une recherche de minimum sur les clés.

```
1 def tourneeMinimale(Population):
2     cle_min = float('inf')
3     for cle in Population:
4         ... # à compléter
5         ... # à compléter
6     return Population[cle_min]
```

(L'expression `float('inf')` représente $+\infty$, un nombre flottant plus grand que tout autre nombre.)

Q8. Compléter les lignes 4 et 5 du code ci-dessus.

b) Reproduction

Il s'agit à partir de deux individus (deux tournées) `parent1` et `parent2` de les croiser afin d'obtenir une nouvelle tournée `enfant`. Pour cela, on choisit aléatoirement un entier k entre 0 et $N-1$, et on complète le préfixe `parent1[:k]` de longueur k de `parent1` par les éléments du suffixe `parent1[k:]` mais ajoutés dans l'ordre où ils apparaissent dans `parent2`. Par exemple avec $N = 9$ et $k = 5$:

parent1 :	**3**	**7**	**2**	**8**	**4**	1	9	6	5
parent2 :	2	**6**	3	4	**1**	8	**5**	7	**9**

$\longmapsto$ enfant :

3	**7**	**2**	**8**	**4**	6	1	5	9

```
1 def croisement(parent1,parent2):
2     k = randint(0,N-1)
3     enfant_debut = parent1[:k]
4     genes_restants = parent1[k:]
5     enfant_fin = []
6     for x in ... :     # à compléter
7         if x in ... : # à compléter
8             ...       # à compléter
9     enfant = enfant_debut + enfant_fin
10    return enfant
```

Le tirage au sort se fait à l'aide de la fonction `randint` du module `random` (qu'on aura importée).

Q9. Compléter les lignes 6 à 8 du code ci-dessus de la fonction `croisement`.

Pour plus de symétrie, chaque couple de parents donnera naissance à deux enfants, par ce procédé, mais en échangeant le rôle des deux parents.

On procède aussi avec une probabilité faible à une mutation aléatoire. Pour cela, on tire au sort deux entiers a et b entre 0 et $N-1$ et on échange dans la liste `enfant` les éléments aux indices a et b.

```
1 def mutation(enfant):
2     a = randint(0,N-1)
3     b = randint(0,N-1)
4     ... # à compléter
```

Q10. Compléter la ligne 4 du code ci-dessus de la fonction `mutation`.

c) Code de l'algorithme génétique

Q11. Écrire la ligne de code permettant d'importer les 3 fonctions `shuffle`, `randint` et `random` du module `random`.

La fonction `Algorithme_Genetique` prend en argument le nombre de générations `NGen`, et les entiers `Np` et `Ns`. Elle renvoie la tournée de longueur minimale parmi tous les individus engendrés.

La probabilité de mutation a été choisie à 0,2.

```
1 def Algorithme_Genetique(NGen, Np, Ns):
2     P0 = PopulationInitiale(Np)           # Population initiale
3     PopulationParent = selection(P0,Ns)   # Population sélectionnée
4     for generations in range(NGen):       # Générations suivantes
5         ListeCles = [ cle for cle in PopulationParent ]
6         n = len(ListeCles)
7         PopulationEnfants = {}            # Génération suivante
8         for couples in range(Np):         # Reproduction
9             p1 = randint(0,n-1)           # choix de deux parents
10            p2 = randint(0,n-1)
11            parent1 = PopulationParent[ListeCles[p1]]
12            parent2 = PopulationParent[ListeCles[p2]]
13            for descendants in range(2): # Obtention de deux enfants
14                enfant = ...              # à compléter
15                if random() < 0.2:
16                    ...                   # à compléter
17                PopulationEnfants[longueur(enfant)] = enfant
18                parent1, parent2 = parent2[:], parent1[:]
19        PopulationParent.update(PopulationEnfants) # Ajout génération suivante
20        PopulationParent = ...            # à compléter # Sélection suivante
21    return ...                            # à compléter
```

À la ligne 19 on utilise la méthode des dictionnaires `dict1.update(dict2)` qui fusionne le dictionnaire `dict2` dans le dictionnaire `dict1`.

Q12. Compléter les lignes 14, 16, 20, 21 par l'une des expressions suivantes :

a) `tourneeMinimale(PopulationParent)` **b)** `croisement(parent1,parent2)`

c) `mutation(enfant)` **d)** `selection(PopulationParent, Ns)`

Par exemple, plusieurs appels de `Algorithme_Genetique(1000,100,80)` permettent d'obtenir après quelques secondes une tournée :
`[11, 12, 2, 6, 17, 5, 4, 9, 19, 18, 10, 14, 15, 20, 1, 16, 8, 7, 13, 3]` de longueur 177.
Tandis que la stratégie naïve évoquée question 2 renvoie la tournée :
`[11, 12, 2, 6, 17, 5, 4, 9, 19, 1, 16, 3, 13, 7, 8, 14, 10, 18, 20, 15]` de longueur 218 et que des tournées choisies au hasard ont des longueurs autour de 500.

Notre stratégie donne de très bons résultats en temps raisonnable.

FIN DU SUJET

Corrigé

Q1. La longueur de la tournée (0,1,3,2,0) est

$$D_{0,1} + D_{1,3} + D_{3,2} + D_{2,0} = 30 + 35 + 22 + 21 = 108$$

et la longueur de la tournée (0,2,3,1,0) est

$$D_{0,2} + D_{2,3} + D_{3,1} + D_{1,0} = 21 + 22 + 35 + 30 = 108.$$

Q2. En suivant la stratégie proposée, on obtient la tournée (0,2,1,3,0). La longueur de cette tournée est

$$D_{0,2} + D_{2,1} + D_{1,3} + D_{3,0} = 21 + 17 + 35 + 25 = 98.$$

Cette tournée n'est pas minimale puisque la tournée (0,1,2,3,0) est plus courte.

Q3. La valeur de `D[1][2]` est 13 et les villes 19 et 20 sont distantes de 10 km.

Q4. On peut compléter le code ainsi :

```
def longueur(tournee):
    lgr = D[0][tournee[0]]
    for k in range(1,N-1):
        lgr = lgr + D[tournee[k]][tournee[k+1]]
    lgr = lgr + D[tournee[N-1]][0]
    return lgr
```

Q5.1. On ne peut pas prendre pour clé les tournées car une clé est un objet non mutable et ne peut donc pas être une liste.

Q5.2. Si on écrivait `T1=T0` plutôt que `T1=T0[:]`, on aurait un phénomène d'aliasing. Ainsi, toutes les clés du dictionnaire seraient associées à une seule et même valeur et cette valeur serait modifiée à chaque tour de boucle.

Q6. La fonction correcte est la fonction `triCroissant_1`.

Q7. On peut compléter la fonction ainsi :

```
def selection(Population, Ns):
    MoindresLongueurs = [ cle for cle in Population ] # Liste des clés
    triCroissant(MoindresLongueurs) # Tri croissant de liste des clés
    MoindresLongueurs = MoindresLongueurs[:Ns] # Sélection des Ns moindres
    Dict = {} # Constitution du dictionnaire
    for cle in MoindresLongueurs:
        Dict[cle] = Polulation[cle]
    return Dict
```

Q8. On peut compléter la fonction ainsi :

```
def tourneeMinimale(Population):
    cle_min = float('inf')
    for cle in Population:
        if cle < cle_min
          cle_min = cle
    return Population[cle_min]
```

Q9. On peut compléter la fonction ainsi :

```
def croisement(parent1,parent2):
    k = randint(0,N-1)
    enfant_debut = parent1[:k]
    genes_restants = parent1[k:]
    enfant_fin = []
    for x in genes_restants:
        if x in parent2:
            enfant_fin.append(x)
    enfant = enfant_debut + enfant_fin
    return enfant
```

Q10. On peut compléter la fonction ainsi :

```
def mutation(enfant):
    a = randint(0,N-1)
    b = randint(0,N-1)
    enfant[a], enfant[b] = enfant[b],enfant[a]
```

Q11. On peut importer les fonctions `shuffle`, `randint` et `random` par le ligne de code suivante :

```
from random import shuffle, randint, random
```

Q12. Le code complet de la fonction `Algorithme_Genetique` est le suivant :

```
def Algorithme_Genetique(NGen, Np, Ns):
    P0 = PopulationInitiale(Np)          # Population initiale
    PopulationParent = selection(P0,Ns)  # Population sélectionnée
    for generations in range(NGen):      # Générations suivantes
        ListeCles = [ cle for cle in PopulationParent ]
        n = len(ListeCles)
        PopulationEnfants = {}           # Génération suivante
        for couples in range(Np):        # Reproduction
            p1 = randint(0,n-1)          # choix de deux parents
            p2 = randint(0,n-1)
            parent1 = PopulationParent[ListeCles[p1]]
            parent2 = PopulationParent[ListeCles[p2]]
            for descendants in range(2): # Obtention de deux enfants
                enfant = croissement(parent1,parent2)
                if random() < 0.2:
                    mutation(enfant)
                PopulationEnfants[longueur(enfant)] = enfant
                parent1, parent2 = parent2[:], parent1[:]
        PopulationParent.update(PopulationEnfants) # Ajout génération suivante
        PopulationParent = selection(PopulationParent, Ns)
    return tourneeMinimale(PopulationParent)
```