Initiation à l'informatique

TD noté, rattrapage 2010

Le programme construit au fur et à mesure des questions devra être imprimé à la fin du TD et rendu. A chaque question correspond une fonction à écrire. Le choix des paramètres et du résultat est laissé libre bien que l'énoncé propose des suggestions. Tout document autorisé.

On souhaite au cours de cette séance traverser quelques villes de France le plus rapidement possible. Il est plutôt évident que le chemin illustré par le graphique 1 n'est pas le plus rapide. On cherchera à implémenter quelques astuces qui permettront de construire un chemin "acceptable".

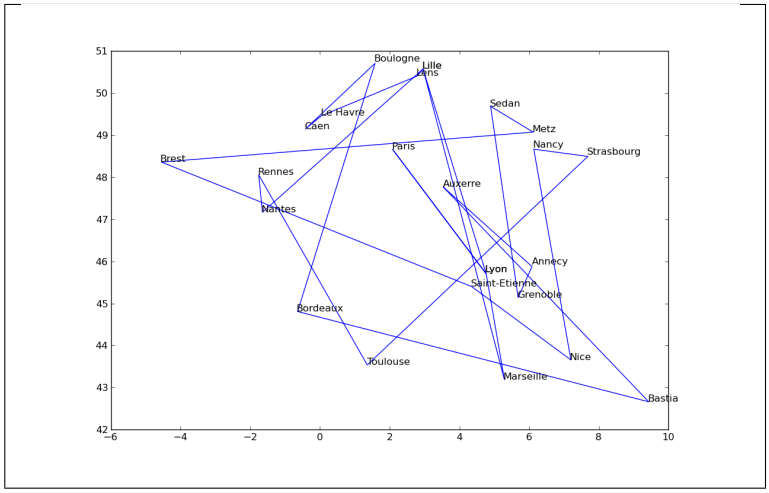

Figure 1 : *Le tour de France. On veut trouver le plus court chemin passant par toutes les villes. Ce problème est aussi connu sous le nom du problème du voyageur de commerce.*

1) La première étape consiste à représenter la liste des villes et de leurs coordonnées via des listes *Python* :

Auxerre	3,537	47,767
Bastia	9,434	42,662
Bordeaux	-0,643	44,808
Boulogne	1,580	50,709
...	...	...
Grenoble	5,684	45,139
Annecy	6,082	45,878

Elles sont accessibles depuis l'adresse http://www.xavierdupre.fr/enseignement/examen_python/villes.txt. Il s'agit de créer une fonction qui récupère ces informations, soit depuis un fichier texte, soit elles peuvent être directement insérées dans le programme sous la forme d'une seule chaîne de caractères.

L'objectif est ensuite d'obtenir une matrice avec le nom de chaque ville en première colonne, l'abscisse et l'ordonnée en seconde et troisième colonnes. Les fonctions `strip`, `replace`, `split` pourraient vous être utiles.

```
[['Auxerre', 3.537309885, 47.767200469999999],
 ['Bastia', 9.4343004229999998, 42.661758419999998],
 ...
```

L'abscisse et l'ordonnée doivent être des réels (`float`) afin d'être facilement manipulées par la suite.

(3 points) *Insérer directement dans le programme la matrice dans sa forme finale ne rapporte pas de point.*

```
def get_tour () :
    stour = """Auxerre        3,537309885        47,76720047
Bastia        9,434300423        42,66175842
Bordeaux        -0,643329978        44,80820084"""
    ...
    return tour
```

2) Ecrire une fonction `distance` qui calcule la distance euclidienne entre deux villes. On supposera que la distance à vol d'oiseau est une approximation acceptable de la distance entre deux villes. (2 points)

```
def distance (tour, i,j) :
    ...
    return d
```

3) Ecrire une fonction `longueur_tour` qui retourne la longueur d'un circuit. Un circuit est décrit par la matrice de la première question : on parcourt les villes les unes à la suite des autres dans l'ordre où elles apparaissent dans la matrice. On commence à Auxerre, on va à Bastia puis Bordeaux pour terminer à Annecy et revenir à Auxerre. (2 points)

```
def longueur_tour (tour) :
    ...
    return d
```

4) Il est facile de vérifier visuellement si un chemin est absurde comme celui de la figure 1. La fonction suivante vous aidera à tracer ce chemin. Il faut la compléter. (2 points)

```
import pylab
def graph (tour) :
    x = [ t[1] for t in tour ]
    y = [ t[2] for t in tour ]
    ....
    ....
    pylab.plot (x,y)
    for ville,x,y in tour :
        pylab.text (x,y,ville)
    pylab.show ()
```

5) La première idée pour construire un chemin plus court est de partir du chemin initial. On échange deux villes choisies aléatoirement puis on calcule la distance du nouveau chemin. Si elle est plus courte, on conserve la modification. Si elle est plus longue, on annule cette modification. On continue tant qu'il n'est plus possible d'améliorer la distance. (4 points)

```
def permutation (tour) :
```

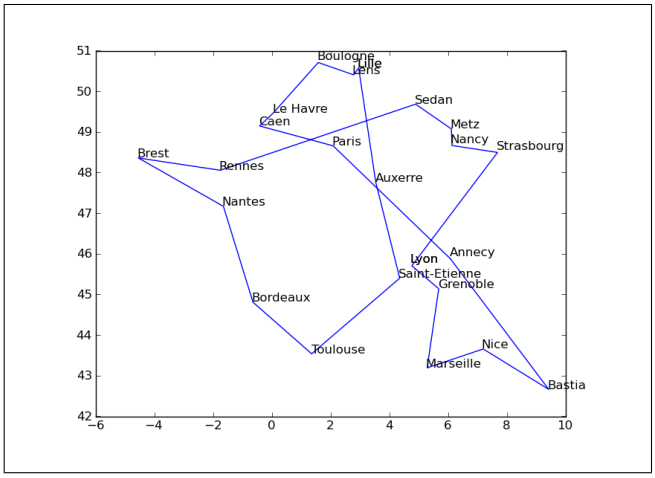

Figure 2 : *Le tour de France lorsque des chemins se croisent. Ce chemin n'est pas optimal de manière évidente.*

6) Le résultat n'est pas parfait. Parfois, les chemins se croisent comme sur la figure 2. Pour cela on va essayer de retourner une partie du chemin. Il s'agit ici de construire une fonction `retourne` qui retourne un chemin entre deux villes i et j. (3 points)

Avant l'exécution de cette fonction, on a :

$$v_1 \to v_2 \to \ldots \to v_{i-1} \to v_i \to v_{i+1} \to \ldots \to v_{j-1} \to v_j \to v_{j+1} \to \ldots$$

Après l'exécution de cette fonction, on a :

$$v_1 \to v_2 \to \ldots \to v_{i-1} \quad v_i \leftarrow v_{i+1} \leftarrow \ldots \leftarrow v_{j-1} \leftarrow v_j \quad v_{j+1} \to \ldots$$

Ou encore :

$$v_1 \to v_2 \to \ldots \to v_{i-1} \to v_j \to v_{j-1} \to \ldots \to v_{i+1} \to v_i \to v_{j+1} \to \ldots$$

```
def retourne (tour, i,j) :
```

7) De la même manière qu'à la question 5, on choisit deux villes au hasard. On applique la fonction précédente entre ces deux villes. Si la distance est plus courte, on garde ce changement, sinon, on revient à la configuration précédente. On répète cette opération tant que la distance du chemin total diminue. (2 points)

```
def croisement (tour) :
```

8) On termine par l'exécution des fonctions `permutation`, `croisement`, `graph`. On vérifie que le chemin obtenu est vraisemblable même s'il n'est pas optimal.

```
def resoud_et_dessine (tour) :
```

Les deux transformations proposées pour modifier le chemin sont décrites par les fonctions `permutation` et `croisement`. A aucun moment, on ne s'est soucié du fait que le chemin est circulaire. Est-ce nécessaire ? Justifier. (2 points)

Correction

La dernière question suggère que l'aspect circulaire du circuit n'a pas été pris en compte par les fonctions `croisement` et `permutation`. Pour échanger deux villes, il n'est nul besoin de tenir compte du fait que la dernière ville est reliée à la première. En ce qui concerne la fonction `croisement`, il est vrai qu'on ne considère aucune portion incluant le segment reliant la première et la dernière ville. Toutefois, lorsqu'on retourne toutes les villes dans l'intervalle $[i]\,j$, on aboutit au même résultat que si on retournait toutes les villes qui n'y sont pas.

```
# coding: latin-1
import random, numpy, math, pylab, copy

###
### réponse à la question 1
###

def get_tour () :
    tour = """Auxerre        3,537309885        47,76720047
Bastia        9,434300423        42,66175842
Bordeaux        -0,643329978        44,80820084
Boulogne        1,579570055        50,70875168
Caen        -0,418989986        49,14748001
Le Havre        0,037500001        49,45898819
Lens        2,786649942        50,40549088
Lille        2,957109928        50,57350159
Lyon        4,768929958        45,70447922
Paris        2,086790085        48,65829086
Lyon        4,768929958        45,70447922
Marseille        5,290060043        43,1927681
Lille        2,957109928        50,57350159
Nantes        -1,650889993        47,16867065
Rennes        -1,759150028        48,05683136
Toulouse        1,356109977        43,5388298
Strasbourg        7,687339783        48,49562836
Nancy        6,134119987        48,66695023
Nice        7,19904995        43,6578598
Saint-Etienne        4,355700016        45,39992905
Brest        -4,552110195        48,36014938
Metz        6,11729002        49,0734787
Sedan        4,896070004        49,68407059
Grenoble        5,684440136        45,13940048
Annecy        6,082499981        45,8782196""".replace (",", ".").split ("\n")
    # ligne d'avant : on découpe l'unique chaîne de caractères

    # ligne suivant : on découpe chaque ligne en colonne
    tour = [ t.strip ("\r\n ").split ("\t") for t in tour ]
    # puis on convertit les deux dernières colonnes
    tour = [ t [:1] + [ float (x) for x in t [1:] ] for t in tour ]
    return tour

###
### réponse à la question 2
###

def distance (tour, i,j) :
    dx = tour [i][1] - tour [j][1]
    dy = tour [i][2] - tour [j][2]
```

```
    return (dx**2 + dy**2) ** 0.5

###
### réponse à la question 3
###

def longueur_tour (tour) :
    # n villes = n segments
    d = 0
    for i in xrange (0,len(tour)-1) :
        d += distance (tour, i,i+1)
    # il ne faut pas oublier de boucler pour le dernier segment
    d += distance (tour, 0,-1)
    return d

###
### réponse à la question 4
###

def graph (tour) :
    x = [ t[1] for t in tour ]
    y = [ t[2] for t in tour ]
    x += [ x [0] ]   # on ajoute la dernière ville pour boucler
    y += [ y [0] ]   #
    pylab.plot (x,y)
    for ville,x,y in tour :
        pylab.text (x,y,ville)
    pylab.show ()

###
### réponse à la question 5
###

def permutation (tour) :

    # on calcule la longueur du tour actuelle
    best  = longueur_tour (tour)

    # variable fix : dit combien d'échanges ont eu lieu depuis la
    # dernière amélioration
    fix   = 0
    while True :
        # on tire deux villes au hasard
        i = random.randint (0, len(tour)-1)
        j = random.randint (0, len(tour)-1)
        if i == j : continue

        # on les échanges si i != j
        e = tour [i]
        tour [i] = tour [j]
        tour [j] = e

        # on calcule la nouvelle longueur
        d = longueur_tour (tour)

        if d >= best :
            # si le résultat est plus long --> retour en arrière
            # ce qui consiste à échanger à nouveau les deux villes
            fix += 1
```

```
            e = tour [i]
            tour [i] = tour [j]
            tour [j] = e
        else :
            # sinon, on garde le tableau tel quel
            best = d
            # et on met fix à 0 pour signifier qu'une modification a eu lieu
            fix = 0

        # si aucune modification n'a eu lieu durant les dernières 10000 itérations,
        # on s'arrête
        if fix > 10000 : break

###
### réponse à la question 6
###

def retourne (tour, i,j) :
    """
    on échange les éléments i et j
    puis i+1 et j-1
    puis i+2 et j-2
    tant que i+k < j-k
    """
    while i <= j :
        e = tour [i]
        tour [i] = tour [j]
        tour [j] = e
        i += 1
        j -= 1

###
### réponse à la question 7
###

def croisement (tour) :
    """
    cette fonction reprend le même schéma que la fonction permutation
    on annule une modification en appelant à nouveau la fonction retourne
    """
    best  = longueur_tour (tour)
    fix   = 0
    while True :
        i = random.randint (0, len(tour)-2)
        j = random.randint (i+1, len(tour)-1)
        retourne (tour, i,j)
        d = longueur_tour (tour)
        if d >= best :
            # retour en arrière
            fix += 1
            retourne (tour, i,j)
        else :
            fix = 0
            best = d
        if fix > 10000 : break

###
### réponse à la question 8
###
```

```
def enchaine (tour) :
    """
    cette fonction est plus complexe que le résultat demandé pour cette question
    on enchaîne les deux fonctions (croisement, permutation) tant que
    la longueur du circuit diminue

    et si jamais cette longueur ne diminue plus, on perturbe le circuit
    au plus deux fois
    en échangeant trois couples de villes choisies au hasard,
    cette dernière partie n'était pas prévue dans l'énoncé
    """
    best = longueur_tour (tour)
    tttt = copy.deepcopy (tour)
    print "debut", best
    nom  = 0
    while True :

        croisement (tour)
        d = longueur_tour (tour)
        print "croisement", d, best

        permutation (tour)
        d = longueur_tour (tour)
        print "permutation", d, best

        if d < best :
            best = d
            tttt = copy.deepcopy (tour)
            nom  = 0
        elif nom > 2 :
            break
        else :
            nom += 1
            for k in range (0,3) :
                i = random.randint (0, len(tour)-2)
                j = random.randint (i+1, len(tour)-1)
                e = tour [i]
                tour [i] = tour [j]
                tour [j] = e

    return tttt

if __name__ == "__main__" :
    tour = get_tour ()
    tour = enchaine (tour)
    graph (tour)
```