Informatique PCSI
Prérequis TP 6 : algorithmes gloutons

Un problème d'optimisation a deux caractéristiques : une fonction que l'on doit maximiser ou minimiser et une série de contraintes auxquelles il faut satisfaire. On peut essayer de résoudre ce type de problème en écrivant un algorithme qui énumère les possibilités de manière exhaustive afin de trouver la meilleure. C'est un algorithme très simple mais souvent inutilisable en machine à cause de son coût. L'objectif d'un algorithme glouton (en anglais « greedy algorithm ») est d'obtenir une solution rapidement. Mais celle-ci n'est pas toujours la solution optimale.

Un choix peut être globalement optimal, c'est le meilleur de tous, ou localement optimal, c'est le meilleur parmi un ensemble restreint de choix. À chaque étape exécutée par un algorithme, se présente un ensemble de choix et un algorithme glouton fait le meilleur choix parmi les propositions. Un choix glouton est donc un choix localement optimal. La question est de savoir si en faisant une série de choix localement optimaux, on fini par aboutir à une solution optimale. C'est parfois le cas mais pas toujours.

Problème du rendu de monnaie

Pour simplifier nous supposons que nous n'avons que des pièces. Un système de pièces est alors un n-uplet $S = (p_0, p_1, \ldots, p_{n-1})$, où p_i représente la valeur de la pièce d'indice i. Ces valeurs constituent une suite de nombres entiers strictement croissante avec $p_0 = 1$. Si $p_0 > 1$, alors certaines sommes ne peuvent pas être rendues.

Le problème du rendu de monnaie consiste à trouver une liste d'entiers positifs $[x_0, x_1, \ldots, x_{n-1}]$ qui vérifie $x_0p_0 + x_1p_1 + \ldots + x_{n-1}p_{n-1} = r$ où r est la somme à rendre en minimisant la somme $x_0 + x_1 + \ldots + x_{n-1}$, c'est-à-dire le nombre de pièces utilisées.

La condition $x_0p_0 + x_1p_1 + \ldots + x_{n-1}p_{n-1} = r$ est appelée contrainte.

Dans le système de la zone euro, par exemple, nous avons en centimes les pièces suivantes :
$S = (1, 2, 5, 10, 20, 50, 100, 200, 500)$ où 100, 200 et 500 représentent respectivement les pièces de 1, 2 et 5 euros.

Il y a de nombreuses manières de rendre huit centimes. Les pièces qui peuvent être utilisées sont les pièces de 1, 2 et 5 centimes. Nous n'écrivons que les triplets possibles : $[8, 0, 0]$, $[6, 1, 0]$, $[4, 2, 0]$, $[3, 0, 1]$, $[2, 3, 0]$, $[1, 1, 1]$ et $[0, 4, 0]$.

Un programme permet de tester de manière exhaustive tous les triplets :

```
p = (1, 2, 5)
for i in range(9):
    for j in range(5):
        for k in range(2):
            s = i * p[0] + j * p[1] + k * p[2]
            if s == 8:
                print([i, j, k])
```

Le triplet qui minimise le nombre de pièces est le triplet $[1, 1, 1]$ avec 3 pièces.

Avec le système donné en exemple, un algorithme glouton fournit la solution optimale. On cherche, par valeur décroissante en partant de la pièce qui a la plus forte valeur, la première pièce qui a une valeur inférieure ou égale à la somme à rendre r. On prend cette pièce, on retranche sa valeur v à r. On recommence en partant de la pièce prise en cherchant celle qui a une valeur inférieure ou égale à la nouvelle somme à rendre $r - v$. On prend cette pièce, et ainsi de suite jusqu'à arriver à une somme à rendre nulle.

Les entrées sont `p` pour le système de pièces et `r` pour la somme à rendre.

```
def monnaie(p, r):
    n = len(p)
    i = n - 1
    solution = n * [0]
    while r > 0:
        while p[i] > r:
            i = i - 1
        solution[i] = solution[i] + 1
        r = r - p[i]
    return solution
```

Test de la fonction :

```
>>> p = (1, 2, 5, 10, 20, 50, 100, 200, 500)
>>> monnaie(p, 8)
[1, 1, 1, 0, 0, 0, 0, 0, 0]
```

Dans le cas général, avec un système différent de celui montré en exemple, c'est un problème difficile à résoudre. Mais dans presque tous les systèmes de monnaie, l'algorithme glouton est optimal. Pour rendre la monnaie, on rend la pièce (ou le billet) de valeur maximale, et on continue tant qu'il reste quelque chose à rendre.

Le système monétaire utilisé en Angleterre jusque dans les années 1960 comportait une multitude de pièces : des pièces de 1 penny, 3 pence, 4 pence, 6 pence, 12 pence soit 1 shilling, etc. (Le mot penny est le singulier de pence.)

Pour rendre 8 pence par exemple, le choix glouton donne $[2, 0, 0, 1]$, soit une pièce de 6 pence et deux pièces d'un penny, alors que le choix optimal est $[0, 0, 2, 0]$, soit 2 pièces de 4 pence.

Plus court chemin dans le plan

Supposons que nous disposons de n points placés dans le plan muni d'un repère orthonormé. Ces points sont donnés par leurs coordonnées. Étant donné un point P quelconque du plan, la question est de trouver un chemin qui commence en P et qui passe par les n points sans jamais passer deux fois par le même point. Un chemin entre deux points est un segment. Quel est le plus court chemin ?

On pourrait envisager de tester tous les chemins possibles et de retenir le plus court. Ce type de méthode s'appelle méthode par « force brute ». Avec trois points A, B, C, nous avons six chemins possibles : P - A - B - C, P - A - C - B, P - B - A - C, P - B - C - A, P - C - A - B, P - C - B - A. Avec 4 points, nous avons 4 fois plus de chemins, soit 24. Avec 10 points nous avons $10 \times 9 \times \ldots \times 2 \times 1 = 3628800$ chemins possibles. Il faut écrire tous ces chemins et déterminer le plus court en faisant autant de tests que de chemins. Admettons que cela puisse se faire en une secondes. Avec 15 points, il y a 1307674368000 chemins possibles, soit environ 3×10^6 fois plus de chemins qu'avec 10 points. Il faudrait donc 3×10^6 secondes pour trouver le plus court, soit plus de 4 jours. Avec 16 points, il y a 16 fois plus de chemins et il faudrait donc plus de 64 jours, et ainsi de suite. On comprend bien que cette méthode n'est utilisable qu'avec un nombre restreint de points. Une méthode gloutonne par contre est envisageable.

Nous supposons que les n points appartiennent à un carré dont la longueur du côté est stockée dans une variable `dim`.

Nous disposons de la distance euclidienne dans le plan et commençons par définir une fonction qui renvoie la distance entre deux points. Un point est représenté par une liste contenant l'abscisse et l'ordonnée du point.

```
from math import sqrt

def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
```

Nous construisons un tableau contenant les distances entre les différents points considérés et entre chacun de ces points et le point P.

La fonction `distances` prend en paramètre la liste des n points et le point de départ P. Le tableau renvoyé `tab` est une liste de $n+1$ listes de longueurs $n+1$. L'élément `tab[i][j]` est la distance entre les points d'indices `i` et `j`, P ayant pour indice `n`. Ce tableau est bien sûr symétrique : la valeur de `tab[i][j]` est égale à celle de `tab[j][i]`.

```
def distances(pts, dep):
    n = len(pts)
    tab = [(n+1)*[0] for i in range(n+1)]
    for i in range(n):
        for j in range(i):
            tab[i][j] = distance(pts[i], pts[j])
            tab[j][i] = tab[i][j]
        tab[n][i] = distance(dep, pts[i])
        tab[i][n] = tab[n][i]
    return tab
```

Notons que la fonction calcule $n(n+1)/2$ distances et a donc un coût qui est quadratique en fonction de n

Nous disposons des distances entre deux points quelconques. Nous pouvons donc écrire une fonction `longueur` qui prend en paramètres un chemin et un tableau des distances, et qui renvoie la longueur du chemin (qui commence en P).

```
def longueur(chemin, dist):
    d = 0
    id_pt = len(dist) - 1
    for point in chemin:
        d = d + dist[id_pt][point]
        id_pt = point
    return d
```

Nous construisons un chemin en choisissant à chaque étape le point le plus proche de la position courante parmi les points disponibles (ceux par lesquels le chemin n'est pas encore passé). On dit aussi le « plus proche voisin ». Autrement dit nous appliquons une stratégie gloutonne.

Pour cela nous écrivons une fonction `indice` qui prend en paramètres la position courante, le tableau des distances et les indices des points disponibles. La fonction renvoie l'indice du point le plus proche.

Pour avoir la liste des points disponibles on pourrait commencer avec la liste de tous les points et supprimer à chaque étape le point choisi. Mais il est plus simple de construire une liste dont les n éléments sont égaux à `True`, l'indice d'un élément correspondant à l'indice d'un point, et de passer la valeur de l'élément d'indice `i` à `False` quand le point d'indice `i` est choisi.

```
def indice(position, dist, dispo):
    n = len(dist) - 1
    mini = 3 * dim # supérieur à la diagonale du carré
    for i in range(n):
        if dispo[i]:
            d = dist[position][i]
            if d < mini:
                mini = d
                ind = i
    return ind
```

Une dernière fonction prend en paramètre le tableau des distances et construit le chemin.

```
def plus_court(dist):
    n = len(dist) - 1
    chemin = []
    dispo = n * [True]
    position = n
    while len(chemin) < n:
        position = indice(position, dist, dispo)
        chemin.append(position)
        dispo[position] = False
    return chemin
```

Pour tester le programme, nous choisissons un nombre de points et la dimension du carré dans lequel ces points sont choisis au hasard. Une fonction `points` crée la liste des n points tous distincts.

```
from random import randint

nbpoints, dim = 10, 20

def points(n, c):
    liste = []
    while len(liste) < n:
        x = randint(-c, c)
        y = randint(-c, c)
        if [x, y] not in liste:
            liste.append([x, y])
    return liste
```

On crée le point de départ et nous pouvons représenter ces points sur un graphique afin de vérifier que le programme est correct.

```
depart = (randint(-dim, dim), randint(-dim, dim))

import matplotlib.pyplot as plt
```

```
pts = points(nbpoints, dim)
x = [u[0] for u in pts]
y = [u[1] for u in pts]
plt.plot(x, y, "x")
plt.plot(depart[0], depart[1], "+")
plt.show()
```

Le test final permet de déterminer le chemin, de l'afficher dans un graphique et d'afficher la longueur du chemin.

```
tableau = distances(pts, depart)
ch = plus_court(tableau)
print(longueur(ch, tableau))

xliste = [depart[0]] + [pts[k][0] for k in ch]
yliste = [depart[1]] + [pts[k][1] for k in ch]
plt.plot(x, y, "x") # affichage des points
plt.plot(depart[0], depart[1], "+")
plt.plot(xliste, yliste) # affichage du chemin
plt.show()
```

Il reste deux questions auxquelles il faut répondre : cet algorithme glouton nous permet-il d'obtenir le chemin le plus court et quel est son coût ?

La réponse est non à la première question. Prenons trois points $A(0\,;0)$, $B(5,1\,;0,2)$ et $C(10\,;0)$. Soit P de coordonnées $(2,6\,;1)$. Avec l'algorithme du plus proche voisin nous obtenons le chemin P - B - C - A. La longueur de ce chemin est $PB+BC+CA \simeq 2,6+4,9+10 = 17,5$. Mais le plus court chemin est P - A - B - C avec une longueur égale à $PA + AB + BC = 2,8 + 5,1 + 4,9 = 12,8$. En général l'algorithme du plus proche voisin fournit un chemin plus long que le plus court chemin et la différence augmente avec le nombre de points. Il est conseillé de faire un dessin.

Pour la deuxième question, nous évaluons le coût de chaque fonction.

La fonction `distances` construit un tableau de $(n+1)^2$ valeurs et a un coût de l'ordre de n^2. La fonction `indice` comporte une boucle et a un coût linéaire. La fonction `plus_court` comporte une boucle avec à chaque passage un appel à la fonction `indice`, donc son coût est quadratique. En conclusion, cet algorithme a un coût quadratique en fonction de n.