TP7

Algorithmes gloutons

Les problèmes d'optimisation

L'optimisation est une branche des mathématiques cherchant à modéliser, à analyser et à résoudre les problèmes qui consistent à minimiser ou maximiser une fonction sur un ensemble.
Les problèmes d'optimisation classiques sont par exemple :

1. la répartition optimale de tâches suivant des critères précis (emploi du temps avec plusieurs contraintes) ;
2. le problème du rendu de monnaie ;
3. le problème du sac à dos ;
4. la recherche d'un plus court chemin dans un graphe ;
5. le problème du voyageur de commerce.

Résoudre un problème d'optimisation : les algorithmes gloutons

De nombreuses techniques informatiques sont susceptibles d'apporter une solution exacte ou approchée à ces problèmes.

- **La recherche de toutes les solutions**
  La technique la plus basique pour résoudre ce type de problème d'optimisation consiste à énumérer de façon exhaustive toutes les solutions possibles, puis à choisir la meilleure. Cette approche par force brute, impose souvent un coût en temps trop important pour être utilisée.
- **Les algorithmes gloutons**
  Un algorithme glouton (greedy algorithm) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local.
  Au cours de la construction de la solution, l'algorithme résout une partie du problème puis se focalise ensuite sur le sous-problème restant à résoudre.
  La méthode gloutonne consiste à choisir des solutions locales optimales d'un problème dans le but d'obtenir une solution optimale globale au problème.
  - ⋆ Le principal avantage des algorithmes gloutons est leur facilité de mise en œuvre.
  - ⋆ Le principal défaut est qu'ils ne renvoient pas toujours la solution optimale nous le verrons.
  - ⋆ Dans certaines situations dites canoniques, il arrive qu'ils renvoient non pas un optimum mais l'optimum d'un problème

I Problème du rendu de monnaie

On veut programmer une caisse automatique pour qu'elle rende la monnaie avec le nombre minimal de pièces et de billets.
On se donne donc une liste de valeurs de billets ou de pièces, par exemple en euros : `liste = [200,100,50,20,10,5,2,1]`.
Une méthode possible est d'utiliser un l'algorithme dont l'idée est la suivante :

- Imaginons qu'on doive rendre 47 euros.
- On commence par chercher la pièce ou le billet de la plus grande valeur possible inférieure ou égale à la somme à rendre, donc ici 20.
- On déduit cette valeur de la somme à rendre qui devient 27 et on recommence jusqu'à obtenir une somme nulle.

On obtiendra sur cet exemple la liste de rendu de monnaie suivante : `monnaie = [20,20,5,2]`.
Sur cet exemple, il n'est pas difficile de montrer que cette liste est optimale car le nombre total de solutions possibles n'est pas très grand.
Le but de l'exercice est décrire une fonction de rendu de monnaie (en question 4) qui met en oeuvre l'algorithme précédent.
Cette fonction de rendu de monnaie utilisera seulement des listes écrites dans l'ordre décroissant.

1. Écrire une fonction `VerifieDecroissant(L)` qui prend en argument une liste `L` et vérifie que les entrées de liste sont bien écrites dans l'ordre décroissant (au sens large). Cette fonction retournera un booléen.
2. Écrire une fonction `MonMax(L)` qui prend en argument une liste `L` et retourne un couple `[M, imax]` où `M` est le maximum des valeurs de `L` et `imax` est un indice tel que `L[imax]` donne cette valeur maximum.
3. Écrire une fonction `MaxEnTete(L)` qui ne retourne rien mais modifie la liste `L` en échangeant la valeur apparaissant en `L[0]` et la valeur `L[imax]` (avec les notations de la question 2.
4. Écrire une fonction `rendre(valeur, liste)` qui prend en argument un entier `valeur` et une liste d'entiers `liste` (supposée rangée dans l'ordre décroissant) et applique l'algorithme glouton décrit ci-dessus pour renvoyer une liste qu'on notera `monnaie` décomposant l'entier valeur à l'aide des éléments de liste comme dans l'exemple du préambule.
5. (a) On suppose que `valeur = 63` et `liste =[200,100,50,20,2,1]`.
   Quelle sera la liste `monnaie` renvoyée par la fonction `rendre(valeur, liste)`?
   (b) Montrer que cette liste `monnaie` n'est pas optimale.

II Problème du voyageur de commerce - Plus court chemin dans un plan

On fixe n villes notées $V_1, V_2, \ldots$ et une position de départ notée Ω. Le but est de visiter chaque ville dans n'importe quel ordre et de revenir à la ville de départ en utilisant le chemin le plus court.
On suppose qu'on connaît un carré contenant l'ensemble de ces villes et dont l'arête sera stockée dans la variable `arete`.

Exercice 1

1. Dénombrer les chemins possibles
2. Expliquer pourquoi un algorithme consistant à calculer toutes les distances de tous les chemins possibles n'est pas envisageable.
3. Proposer une stratégie plus efficace

Exercice 2

Les coordonnées des différents points du plan sont stockés dans une liste à deux éléments `[x, y]`.
Les différentes coordonnées des villes sont stockées dans une liste. Par exemple la liste `[[1, 2], [-3, 5]]` correspond aux coordonnées de deux villes $V_1(1,2)$ et $V_2(-3,5)$.
Le point de départ Ω n'appartient pas à la liste des coordonnées.

Écrire une fonction `distance(p1,p2)` où `p1` et `p2` sont les coordonnées de deux points et qui renvoie la distance entre ces deux points.

La première étape pour mettre en place la stratégie gloutonne est de calculer les distance entre chaque point (ville ou point de départ). Toutes les distances seront stockée dans un tableau représenté par une liste de liste comme ci-dessus.

```
#distance entre V1(1, 2), V2(-3, 5) et Omega(0, 0)
d =
[[0, 5.0, 2.23606797749977],    # <-- [d[V_1, V_1], d[V_1, V_2], d[V_1, Omega]
[5.0, 0, 5.830951894845301],    # <-- [d[V_2, V_1], d[V_2, V_2], d[V_2, Omega]
[2.23606797749977, 5.830951894845301, 0]]   #<-- [d[Omega, V_1], d[Omega, V_2
], d[Omega, Omega]
```

Exercice 3

1. Compléter le programme ci-dessous permettant de construire le tableau d.

```
def distances(points, omega):
    '''
    Entrée : liste de coordonnées des points, liste des coordonnées de
Omega
    Sortie : liste de liste
    Distances entre tous les points A, B, ..., Omega
    '''
    n = len(points)
    d = [..... * [0] for i in range(.....)]
    for i in range(n):
        for j in range(i):
            d[i][j] = .....
            d[j][i] = .....
    for i in range(n):
        d[i][n] = .....
        d[n][i] = .....
    return d
```

2. Combien de fois la fonction `distance` a-t-elle été appelée?

On continue de mettre en place la stratégie gloutonne : nous construisons un chemin en choisissant à chaque étape la ville le plus proche de notre position parmi les villes disponibles (celles par lesquelles on n'est pas encore passé). On dit aussi « le plus proche voisin ».

Exercice 4

Compléter la fonction `IndiceSuivant` permettant de déterminer l'indice de la ville suivante.

```
def IndiceSuivant(position, d, dispo):
    '''
    Entrée : indice de la position, tableau des distances,
    liste de booléen coorespondant aux indices des villes visitées ou non
    Sortie : indice de la ville suivante
    '''

    n = len(d) - 1
    mini = 2 * arete # La distance entre deux points est inférieure 2 *
arete
    for i in range(n):
        if ..............  :
            # A compléter
    return indice,nonVisite
```

Exercice 5

1. Ecrire la fonction `PlusCourtChemin` prenant comme argument la liste des villes et le point de départ et retournant la liste des indices des villes visitées.
2. Combien d'appels à la fonction `distance` sont effectués ?

III Le problème du choix d'activités

Supposons avoir une liste d'activités, chacune associée à un créneau horaire défini par une heure de début et une heure de fin. Deux activités sont compatibles si leurs créneaux horaires ne se recouvrent pas.
On souhaite sélectionner un nombre maximal d'activités toutes compatibles entre elles.

Exercice 6

1. On se donne des activités avec les créneaux suivants : 8h-13h, 12H-17h, 9h-11H, 14h-16h, 11h-12h.
   Combien de ces activités peuvent-elles être conciliées sur une seule journée ?
2. On propose une stratégie gloutonne pour sélectionner des activités en commençant par le début de la journée : choisir l'activité dont l'heure de fin arrive le plus tôt (parmi les activités dont l'heure de début est bien postérieure aux créneaux des activités déjà choisies).
   Appliquer cette stratégie à la situation précédente.

Exercice 7 (*Programmation de la stratégie gloutonne*)

ON suppose avoir `n` activités notées de `0` à `n - 1`, et deux listes `debut` et `fin` de taille `n` tels que `debut[i]` et `fin[i]` contiennent respectivement l'heure de début et l'heure de fin de l'activité `i`.

1. Écrire une fonction `prochaine(debut, fin, h)` qui sélectionne, parmi les activités dont l'heure de début n'est pas antérieure à `h`, une s'arrêtant le plus tôt. On demandera à la fonction de renvoyer `None` si aucun créneau n'est compatible.
2. En déduire une fonction `selection(debut, fin)` qui, en supposant que toutes les heures sont positives, sélectionne autant d'activités que possible en suivant la stratégie gloutonne. On demandera à la fonction d'afficher les numéros des activités sélectionnées.