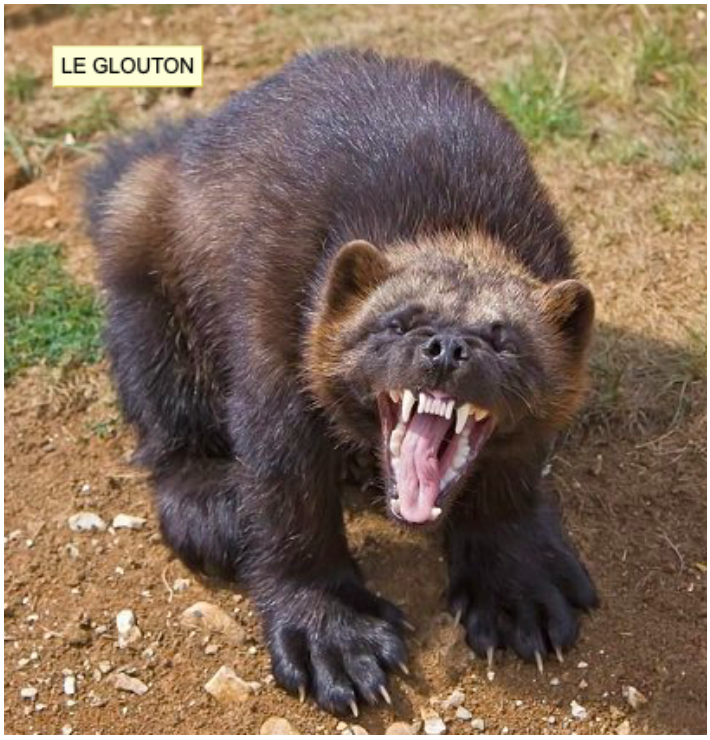
TP INFO 3 | Introduction aux algorithmes gloutons

Dans ce TP, nous allons nous intéresser à des problèmes d'optimisation et à une classe particulière d'algorithmes associés, qualifiés de « gloutons » (greedy algorithms).



1. Exemples introductifs

Les problèmes d'optimisation consistent en la recherche d'un maximum ou d'un minimum.

1.1. Payer avec le moins de pièces possible

Nous commençons par un problème très classique et concret. Parmi toutes les façons de payer une somme de 153 € avec des billets de 5 € et des pièces 1 € et 2 €, celle qui nécessite le moins de devises est clairement la suivante :

$$153 = 30 \times 5 + 1 \times 2 + 1 \times 1$$

Plus généralement, supposons que l'on dispose de p devises de valeurs entières $v_0 > \cdots > v_{p-1}$ avec $v_{p-1} = 1$.

Comment payer une somme de n (un entier naturel non nul) en utilisant le moins de pièces possible ?

Puisque $v_{p-1} = 1$, on peut toujours payer la somme. Une idée naturelle est de commencer par payer le plus possible avec la devise de valeur v_0, puis de recommencer avec la devise de valeur v_1, etc. jusqu'à tout payer. Ce principe est qualifié de glouton car il consiste à faire un choix optimal à chaque itération dans l'espoir d'obtenir une solution optimale globale.

Le principe général est de commencer par payer le plus possible avec v_0 :

$$n = x_0 \times v_0 + r_0 \quad \text{où le reste à payer } r_0 \text{ vérifie } 0 \leqslant r_0 < v_0$$

On reconnaît la division euclidienne de n par v_0. On itère ensuite ces divisions euclidiennes :

$$r_0 = x_1 \times v_1 + r_1 \quad , \quad r_1 = x_2 \times v_2 + r_2 \quad , \quad r_2 = x_3 \times v_3 + r_3 \quad \text{etc.}$$

Cette approche ne se solde pas toujours par une solution optimale. Par exemple, pour trois devises de valeurs 4, 3 et 1, l'algorithme donne pour le paiement de $n = 6$:

$$6 = 1 \times 4 + 0 \times 3 + 2 \times 1$$

alors que l'optimum est clairement $6 = 0 \times 4 + 2 \times 3 + 0 \times 1$.

Bien que ne donnnant pas toujours l'optimum, nous allons coder cet algorithme en Python. On utilise deux variables d'entrée : `n` du type `int` (contenant la valeur de n) et `valeurs` du type `list` (contenant dans cet ordre v_0, …, v_{p-1} avec $v_{p-1} = 1$). On renvoie la solution sous la forme d'une liste `paiement=[x_0,...,x_p-1]` (correspondant à un paiement avec x_i fois la pièce de valeur v_i pour tout $i \in [\![0, p-1]\!]$).

On parcourt les valeurs dans le sens décroissant en mettant à jour la variable `reste` (contenant le reste à payer au fur et à mesure que les valeurs des pièces décroissent) et la variable `rep` qui contiendra la solution en fin d'itération.

```
def monnaie(n,valeurs):
    reste,rep=n,[]
    for i in range(len(valeurs)):
        rep.append(reste//valeurs[i])
        reste=reste%valeurs[i]
    return rep
```

1.2. Recherche d'un chemin de coût minimal dans une matrice

Soit M une matrice réelle à n lignes et n colonnes dont les lignes et les colonnes sont numérotées de 0 à $n-1$. On appelle chemin diagonal dans M toute liste de coefficients de M commençant à $M_{0,0}$ et finissant à $M_{n-1,n-1}$ en se déplaçant à chaque itération soit vers le bas, soit vers la droite :

Par exemple, pour une matrice carrée de taille $n = 5$, vous trouverez tracés ci-contre deux chemins diagonaux. À tout chemin diagonal est associé son coût qui n'est autre que la somme des coefficients du chemin :

38 et 36 pour les chemins ci-contre

$$\begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix} \qquad \begin{pmatrix} 1 & 5 & 2 & 5 & 7 & 9 \\ 7 & 3 & 4 & 1 & 2 & 4 \\ 1 & 0 & 4 & 7 & 2 & 1 \\ 2 & 6 & 2 & 1 & 0 & 5 \\ 0 & 1 & 3 & 8 & 9 & 3 \\ 5 & 0 & 7 & 1 & 2 & 5 \end{pmatrix}$$

L'objectif est de trouver un algorithme permettant de calculer un chemin diagonal de M de coût minimal, pour une matrice M de réels quelconque.

Les matrices seront représentées par des listes de listes. Ainsi, pour notre exemple :

```
[[1,5,2,5,7,9],[7,3,4,1,2,4], [1,0,4,7,2,1], [2,6,2,1,0,5], [0,1,3,8,9,3], [5,0,7,1,2,5]]
```

Un chemin diagonal sera codé par une chaîne (type `str` de Python) formée sur les caractères `"d"` (pour *droite*) et `"b"` (pour *bas*). Par exemple, les chemins ci-dessus sont représentés respectivement par les chaînes suivantes :

`"dbddbbbbdd"` et `"bbbdddbbdd"`

On propose l'algorithme glouton suivant : on part de $M_{0,0}$ puis, à chaque itération, on choisit le déplacement local optimal, i.e. on se dirige vers le coefficient minimal (s'il y a deux déplacements possibles, sinon on n'a pas le choix), jusqu'à arriver au coefficient $M_{n-1,n-1}$.

1 *Recherche d'un chemin de coût minimal dans une matrice* ***ff***

1. Donner une matrice de taille $n = 3$ pour laquelle l'algorithme décrit ci-dessus est correct. Idem avec incorrect.
2. Écrire une fonction `gloumin` d'argument une matrice M renvoyant le chemin obtenu par l'algorithme glouton décrit ci-dessus.
3. Vérifier que l'appel `gloumin(M0)` renvoie le chemin `"ddbddbbdbb"`, où M_0 est la matrice donnée en exemple.

1.3. Principe d'un algorithme glouton

Le principe d'algorithme glouton est d'effectuer *un choix optimal* à chaque étape en espérant obtenir *une solution globale optimale*. Cette approche ne conduit pas toujours à une solution effectivement optimale (cf. les deux exemples introductifs).

Forces et faiblesses des algorithmes gloutons

⇛ Nous avons vu qu'un algorithme glouton n'aboutit pas toujours à une solution optimale.

⇛ Néanmoins, les algorithmes gloutons présentent l'avantage d'être faciles à implémenter par rapport à d'autres approches algorithmiques.

2. Exercices

2 — *Des sauts de grenouille f*

Une grenouille se déplace sur des nénuphars en ligne droite et par sauts successifs. On supposera la droite graduée de 0 à $n-1$ (où n est un entier naturel non nul), les nénuphars se trouvant à des abscisses entières (0 pour le nénuphar de départ et $n-1$ pour celui d'arrivée). Voici un exemple, avec $n = 13$:

Connaissant les positions des nénuphars et sachant que la grenouille peut sauter au maximum une distance de r unités d'abscisse, comment celle-ci doit se déplacer du premier au dernier nénuphar de façon à minimiser son nombre total de saut(s) ? Il est assez naturel de penser qu'une solution optimale sera obtenue en faisant à chaque étape un saut de longueur maximal. C'est un algorithme glouton. Pour l'exemple précédent et la valeur $r = 3$, on obtient donc le schéma suivant :

On supposera la configuration codée au moyen de deux variables : `r` de type `int` (contenant la valeur de r) et `pos` de type `list` (contenant les abscisses des nénuphars dans l'ordre croissant, qui commence donc à 0 et finit à $n-1$). La solution renvoyée par l'algorithme sera codée par une liste d'abscisses croissantes de 0 à $n-1$, représentant les différentes étapes de la grenouille.

Sur l'exemple ci-dessus, on aura `r=3`, `pos=[0,3,4,7,8,9,12]` et la réponse sera `[0,3,4,7,9,12]`.

1. On note $v_1, \ldots, v_m$ les abscisses croissantes des nénuphars (avec $v_1 = 0$ et $v_m = n-1$). Donner une condition *nécessaire et suffisante* sur r et les v_i pour que le problème ait une solution.
2. En déduire une fonction `admetSolution` d'arguments `r` et `pos` renvoyant `True` si le problème admet une solution et `False` sinon.
3. Écrire une fonction `saut` d'arguments `r`, `pos` et `i`, renvoyant (en supposant qu'il existe) le plus grand indice `j` tel que `j>i` et `pos[j]-pos[i]<= r`.
4. En utilisant les fonctions `admetSolution` et `saut`, écrire une fonction `sautsDeGrenouille` d'argument `r` et `pos` renvoyant `-1` si le problème n'admet pas de solution et une solution (au format indiqué ci-dessus) dans le cas contraire.

3 *Allocation optimale d'une salle d'examen* ***ff***

Dans une salle d'examen doivent se dérouler une série d'épreuves un jour donné. Ces dernières sont caractérisées par une heure de début d et une heure de fin f, comprises entre 8 et 19. On souhaite planifier le plus possible d'épreuves, deux épreuves ne pouvant avoir lieu en même temps (leurs intervalles de temps ouverts doivent être disjoints, elles sont alors dites compatibles). Les différentes épreuves sont supposées stockées sous la forme d'une liste `ep` de listes au format `[d,f]`, triées par heure de fin croissante.

1. Donner une condition nécessaire et suffisante de compatibilité des épreuves `[d1,f1]` et `[d2,f2]`.
2. On adopte *l'heuristique gloutonne* suivante : on choisit l'épreuve se terminant au plus tôt, puis l'épreuve se terminant au plus tôt parmi celles qui sont compatibles avec la première, etc. jusqu'à épuisement des épreuves. Écrire une fonction `allocation(ep)` renvoyant la liste des épreuves résultat de cet algorithme (au format `[i1,...,ip]` où `i1`, ..., `ip` sont les indices dans la liste `ep` des épreuves sélectionnées).
3. On peut démontrer que l'heuristique gloutonne décrite ci-dessus donne toujours le résultat optimal. On suppose dans cette question que les épreuves sont triées par durée croissante et on adopte l'heuristique gloutonne suivante : on choisit la plus courte, puis la plus courte parmi celles qui lui sont compatibles, etc. Ce choix mène-t-il toujours à une solution optimale ?

4 *Allocation de salles de cours* ***fff***

Différents cours ont lieu un jour donné. Ces derniers sont caractérisés par une heure de début d et une heure de fin f, comprises entre 8 et 19. À chaque cours, il faut attribuer une salle. On souhaite trouver un planning optimal de ces cours, ie mobilisant le moins de salle(s) possible. Les différents cours sont supposés stockés sous la forme d'une liste `cours` de listes au format `[d,f]`, triés par heure de début croissante. Deux cours sont dits compatibles s'ils peuvent avoir lieu dans une même salle. Les salles attribuées seront numérotées à partir de 0 (et on supposera qu'elles sont en nombre suffisant).

1. Donner une condition nécessaire et suffisante de compatibilité des cours `[d1,f1]` et `[d2,f2]`.
2. On adopte *l'heuristique gloutonne* suivante : on parcourt les cours dans l'ordre chronologique de leur début et, pour chacun d'entre eux, si on trouve une salle déjà attribuée et possible pour ce cours, sinon on lui attribue une nouvelle salle. Écrire une fonction `allocation(cours)` renvoyant le planning résultant de cet algorithme (au format `[s1,...,sn]` où `si` est le numéro de la salle du cours numéro i).

3. Indications

1

Attention, il faut tenir compte dans la fonction `gloumin` du cas où on arrive sur un des bords de la matrice (dernière ligne ou dernière colonne).

2

Au 4., l'idée est d'itérer la fonction `saut` jusqu'à épuisement des nénuphars.

3

Il suffit de parcourir dans l'ordre croissant des indices la liste `ep` et de retenir une épreuve si elle est compatible avec la dernière épreuve enregistrée.

4

On pourra utiliser une variable `courssalles` de type `list` contenant, pour chaque salle déjà attribuée, le numéro du dernier cours qui y est programmé (cette liste sera utile pour déterminer si on peut à nouveau attribuer une salle à un nouveau cours ou s'il faut en attribuer une nouvelle).