Algorithmes gloutons

Parmi les algorithmes souvent utilisés pour résoudre des problèmes d'optimisation figurent les algorithmes dits gloutons. Leur fonctionnement est de faire des choix locaux optimaux, c'est-à-dire de toujours faire le choix qui semble le meilleur sur le moment, et de ne jamais revenir sur l'un de ces choix.

Dans certains cas, cette approche conduit bel et bien à une solution optimale, mais il est aussi des problèmes pour lesquelles l'approche gloutonne ne donnera qu'une solution approchée. On pourra parler d'heuristique gloutonne dans ce dernier cas.

Bien sûr, on lance spyder, et vous trouverez à l'adresse : `http://cahier-de-prepa.fr/pcsi-bdb/docs?rep=111` un script à compléter. Téléchargez celui-ci et sauvegardez-le dans votre répertoire personnel.

1 Problèmes 18 et 67 du projet Euler

On suppose donné un ensemble d'entiers strictement positifs organisés sur un triangle de n lignes. Voici un exemple avec $n=4$:

$$\begin{array}{ccccccc} & & & \mathbf{3} & & & \\ & & \mathbf{7} & & 4 & & \\ & 2 & & \mathbf{4} & & 6 & \\ 8 & & 5 & & \mathbf{9} & & 3 \end{array}$$

et on souhaite déterminer un chemin conduisant à la somme maximale depuis le sommet jusqu'à la dernière ligne, où on prend un nombre par ligne en se déplaçant ligne à ligne d'une case à gauche ou à droite. Sur l'exemple ci-dessus, un chemin optimal est celui qui partant du sommet 3 va d'abord à gauche (vers le 7), puis à droite (vers le 4) et encore à droite (vers le 9).

On se convainc assez vite que ce chemin est non seulement optimal, mais aussi unique. Ce chemin pourrait nous être donné par un algorithme glouton, lequel consiste à chaque étape à choisir la plus grande valeur parmi les deux valeurs possibles.

Question 1. Compléter la fonction d'entête `def euler18glouton(T):` pour réaliser ce travail, sachant que `T` représente notre triangle de nombres sous la forme d'une liste de listes. (Voir les exemples `T1` à `T4` dans le script).

Bien sûr, on teste avec les exemples `T*` présents dans le script.

Indication : on pourra noter que, partant du nombre `T[i][j]`, celui situé immédiatement à sa gauche et à la ligne suivante est `T[i+1][j]` et celui situé immédiatement à sa droite et à la ligne suivante est `T[i+1][j+1]`.

On s'en rend vite compte, il est des exemples où l'algorithme glouton ne donne pas la réponse optimale attendue, ce qui au fond était un peu prévisible sur un tel problème (voir `T2` par exemple). On peut tout de même en donner une réponse correcte assez simplement, à l'aide d'une fonction récursive très simple :

Question 2. La fonction `euler18rec(T, i, j):` du script proposé prend en argument notre triangle de nombres, toujours sous la forme d'une liste de listes, et la position d'un des nombres de celui-ci, d'indice de ligne `i` et d'indice de colonne `j`. Son objet est de renvoyer la plus grande somme obtenue à partir de cette position.

Par exemple, `T1` désignant le triangle de nombres présenté ci-dessus, `euler18rec(T1, 1, 0)` devra renvoyer $20(=7+4+9)$ tandis que `euler18rec(T1, 1, 1)` renverra plutôt $19(=4+6+9)$.

Compléter alors la fonction d'entête `def euler18rec(T, i, j):` pour qu'elle réponde aux spécifications attendues. (L'idée est que si on connaît les valeurs renvoyées par `euler18rec(T, i+1, j)` et `euler18rec(T, i+1, j+1)`, alors il n'est pas bien difficile d'en déduire `euler18rec(T, i, j)`.)

Question 3. Tester avec `T1, T2, T3` et `T4`, depuis la position initiale (`i=0` et `j=0`), et comparer avec les réponses obtenues par l'algorithme glouton.

Question 4. Que dire du temps d'exécution pour `T4` ? Comment expliquer un temps si long ? De quel ordre de grandeur serait selon vous le temps d'exécution, depuis la position initale, pour un triangle de n lignes ?

(Pour les plus rapides d'entre-vous, ou en recherche à la maison, des méthodes pour améliorer la complexité de cet algorithme vous sont proposées à la fin du TP)

2 Sélection d'activités

Soit un ensemble S de n d'activités concurrentes (qu'on numérote ici de 1 à n), connaissant pour chacune un horaire de début d_i et un horaire de fin f_i. Les activités i et j sont compatibles si et seulement si $[d_i, f_i[$ et $[d_j, f_j[$ sont disjoints. On cherche à effectuer le maximum d'activités compatibles entre elles.

L'algorithme envisagé est le suivant :

- On suppose triées les activités par heures de fin croissantes $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$.
- La première activité est dès lors automatiquement sélectionnée, puis on passe en revue les autres, dans l'ordre ci-dessus.
- A chaque activité envisagée, elle est sélectionnée si elle est compatible avec la dernière sélectionnée.

Question 5. On suppose ici $n = 9$ et les horaires des activités constituent la liste (triée par heure de fin) :

```
[[1, 4], [3, 5], [0, 6], [3, 7], [5, 7], [5, 9], [7, 10], [9, 12], [11, 14]]
```

Quelles sont les activités sélectionnées par l'algorithme précédent ?

Question 6. Compléter la fonction d'entête `def selectionActivités(Act):` du script fourni pour mettre en œuvre cet algorithme. On testera avec la liste `Act` définie dans le script bien sûr (laquelle n'est autre que celle de la question précédente)

2.1 Preuve de l'optimalité de l'algorithme présenté

On suppose ici que S est formé de n activités, et que celles-ci, d'horaires d_i et f_i pour $1 \leqslant i \leqslant n$ sont triées par heures de fin croissantes. On suppose encore que $\{i_1, \cdots, i_p\}$ où $i_1 < \cdots < i_p$ est une solution optimale, et que $\{j_1, \cdots, j_q\}$ est l'ensemble des activités obtenu par l'algorithme glouton présenté ci-dessus (bien sûr, j_1 ne peut valoir que 1). Il s'agit d'établir que $q \geqslant p$ (et forcément alors $q = p$)

Première remarque : $f_{j_1} \leqslant f_{i_1}$ puisque la première activité sélectionnée par l'algorithme glouton est celle qui se termine en premier, ce qui indique que les activités d'indices $j_1, i_2, \cdots, i_p$ sont encore compatibles et $\{j_1, i_2, \cdots, i_p\}$ est encore une solution optimale. Quitte à remplacer i_1 par j_1, on suppose désormais que $i_1 = j_1$.

Si maintenant on retire de S toutes les activités incompatibles avec j_1 (toutes celles donc qui commencent avant f_{j_1}), $\{i_2, \cdots, i_p\}$ en est alors un solution optimale (qui maximise le nombre d'activités compatibles entre elles). En effet, il s'agit bien sûr d'activités compatibles entre elles, et si on pouvait faire mieux, on trouverait aussitôt une solution meilleure pour l'ensemble S initial que celle qu'on avait supposée optimale.

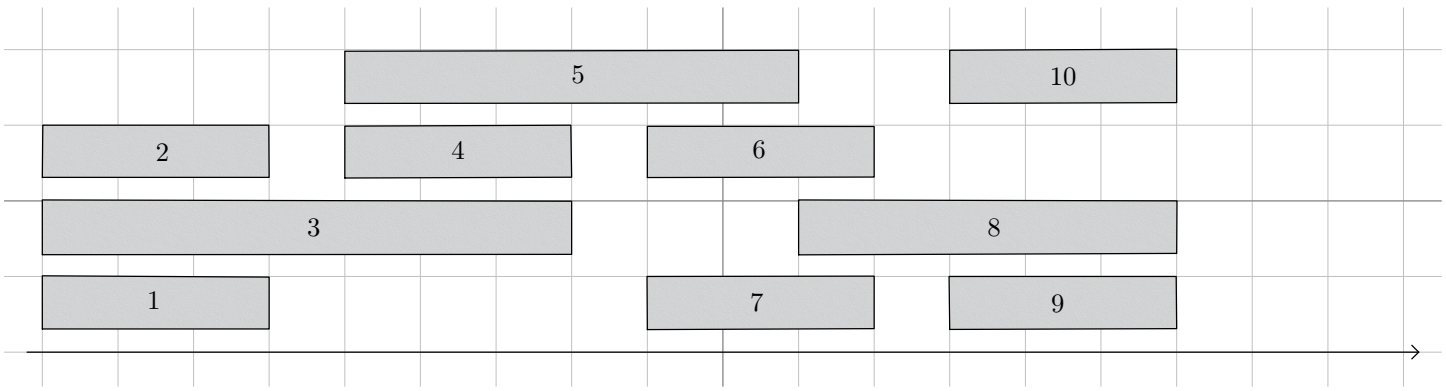
On peut alors remplacer de la même manière i_2 par j_2 et ainsi de suite ce qui établit que $q \geqslant p$. De l'optimalité de la solution initiale, on déduit bien sûr que $q = p$ et que donc l'algorithme glouton fournit bien une solution optimale au problème de sélection d'activités.

3 Allocation de salles pour des cours

Etant donné un certain nombre de cours, mettons n, dont l'horaire de début et de fin est fixé pour chacun (d_j et f_f, qu'on supposera entiers et tels que $d_j < f_j$, on considèrera que le cours a lieu sur l'intervalle de temps $[d_j, f_j[$).

L'objectif est d'allouer une salle pour chaque cours en minimisant le nombre de salles mobilisées pour l'ensemble des n cours. Une solution mobilisant p salles est valable bien sûr pourvu que deux cours ne sauraient, même sur une courte durée, avoir lieu dans la même salle, et optimale si on ne saurait trouver une autre solution mobilisant strictement moins que p salles.

Exemple : voici une organisation en 4 salles de 10 cours :



Question 7. L'organisation proposée ci-dessus est-elle optimale ?

Un algorithme glouton pour décider d'une organisation de ces cours est le suivant :

- On suppose triés les cours selon l'heure de début d_j (dans le sens croissant) (la numérotation des cours de l'exemple précédent respecte cette propriété)
- On alloue une salle pour le cours $[d_1, f_1[$, puis pour le cours $[d_2, f_2[$ et ainsi de suite jusqu'à $[d_n, f_n[$.

  Initialement, aucune salle n'est nécessaire.
- A chaque nouveau cours j, on regarde si parmi les salles introduites précédemment, l'une d'elle est libre à partir de d_j. Si oui, le cours j sera programmé dans cette salle.

  Si aucune salle n'est libre à partir de d_j, on introduit alors une nouvelle salle et on y programme le cours j.

Question 8. A la main, que donnerait l'algorithme glouton ainsi décrit sur l'exemple vu plus haut ?

Question 9. On va implémenter l'algorithme glouton pour ce problème. On construira au fur et à mesure deux listes : l'une associant à un cours donné la salle qui lui est attribuée, et l'autre gardant trace au fur et à mesure de l'allocation de salles de l'heure de fin du dernier cours programmé pour chaque salle. Compléter alors la fonction d'entête `def allocationSalles(Cours):` du script fourni.

On testera sur l'exemple présenté ci-dessus (la liste `Cours` est dans le script fourni)

3.1 Preuve de l'optimalité de l'algorithme glouton pour l'allocation de salles

Il suffit de se rendre compte qu'à chaque fois qu'une nouvelle salle est introduite par notre algorithme on a la certitude qu'aucune autre organisation des cours ne ferait mieux : si pour le cours numéroté j auquel on cherche à attribuer une salle aucune des p salles utilisées jusque-là n'est libre, c'est que parmi les $j-1$ cours qui commencent avant ou en même temps que le cours j et auxquels une salle a été attribuée, il y en a p qui s'avèrent incompatibles ($d_j \in [d_i, f_i[$) avec le cours j ce qui indique qu'il y a manifestement parmi les j premiers cours $p+1$ d'entre-eux qui au moins en partie (sur $[d_j, d_j+1[$) ont lieu en même temps ce qui prouve la nécessité d'au moins $p+1$ salles.

4 Eléments théoriques

Le nombre de problèmes qui peuvent être abordés par un algorithme glouton est tout de même assez important et, lorsque cela est possible, un algorithme glouton est celui qui sera en termes de complexité (en temps comme en mémoire) le plus efficace.

Le premier point qui peut justifier une approche gloutonne est la présence d'une **sous-structure optimale** : cela se dit d'un problème dont une solution optimale contient la solution optimale de sous-problèmes. C'est au fond le cas du premier problème étudié, car en supprimant les premières étapes d'une solution optimale, on obtient encore une solution optimale depuis cette nouvelle position (ligne i et colonne j, d'un parcours optimal donc) jusqu'à la fin.

Pour la sélection d'activités, on l'a vu : si $\{i_1, \cdots, i_p\}$ est une solution optimale, $\{i_2, \cdots, i_p\}$ forme également une solution optimale du sous-problème obtenu en supprimant i_1 et les activités incompatibles avec i_1.

Pour l'allocation de salles pour des cours, on voit que si $[s_1, \cdots, s_n]$ est la solution optimale fournie par l'algorithme glouton où pour tout i, s_i désigne le numéro de salle alloué au cours i, alors $[s_1, \cdots, s_p]$ est également une solution optimale pour le problème posé par les p premiers cours (triés selon l'heure de début).

Une chose est claire : la présence d'une sous-structure optimale n'est pas en soi suffisante, comme le montre le premier problème. C'est juste un indice laissant à penser qu'un algorithme glouton est peut-être approprié, et sinon on pourra mettre en œuvre un algorithme dynamique (voir un peu plus loin pour une solution en programmation dynamique du premier problème, mais l'étude de la programmation dynamique sera reprise plus en détails en seconde année).

Un second point qui, combiné au premier, justifie une démarche gloutonne est la **propriété de choix glouton** : un choix localement optimal peut conduire à une solution (globale) optimale : par exemple pour la sélection d'activités, le choix glouton conduisant à commencer par l'activité portant le numéro 1 (celle qui se termine en premier), la propriété de choix glouton exprime qu'il existe une solution optimale commençant par 1 (ce qu'on a mis en évidence).

Lorsque les deux propriétés de sous-structure optimale de de choix glouton se trouvent réunies, un algorithme glouton est une solution appropriée (et dans ces conditions la solution à privilégier, car la moins coûteuse).

4.1 Exemples de problèmes qui se prêtent à une approche gloutonne

- Le problème du **rendu de monnaie** : pour faire simple, on imagine disposer d'une quantité infinie de pièces de 5, 2 et 1 euros, et on souhaite obtenir une somme d'argent, mettons x euros, en le moins de pièces possible. L'algorithme glouton est celui auquel tout un chacun penserait : d'abord on prend autant de pièces de 5 euros que possible, puis on complète de pièces de 2 euros la somme restante et, si nécessaire, on rajoute une dernière pièce de 1 euro.

Faire la preuve que l'algorithme glouton conduit bien à une solution optimale est ici faisable (et est un excellent exercice !) mais selon les valeurs des pièces disponibles, la stratégie gloutonne peut ne pas être optimale. Un exemple classique est le système monétaire anglais avant 1971 : une livre vaut 20 shillings et un shilling vaut 12 pence (pluriel de penny), et il y avait des pièces de 1 penny, 3 pence, 4 pence et 6 pence. Si on se concentre sur les 3 plus petites pièces (on dispose donc de pièces de 1 penny, 3 pence et 4 pence) et si on veut à l'aide de ces pièces obtenir la somme d'un 1/2 shilling (6 pence), alors la stratégie gloutonne conduit à 3 pièces, ce qui vous l'avez compris n'est pas optimal.

Bref, l'algorithme glouton pour le rendu de monnaie ne conduit à une solution optimale que pour des valeurs adaptées des pièces : on parle de système canonique lorsque la stratégie gloutonne conduit bien, en toutes circonstances, à une solution optimale. (La justification qu'un ensemble de valeurs constitue un système canonique n'est de manière générale pas triviale !)

- On verra un peu plus tard dans l'année l'algorithme de Dijkstra qui cherche un plus court chemin d'un sommet d'un graphe pondéré vers un autre sommet (à vrai dire la même recherche conduit à la détermination du plus court chemin d'un sommet initial à tous les autres sommets du graphe)

5 Pour aller plus loin : retour aux problèmes 18 et 67 du projet Euler

La solution exacte donnée au problème 18 par une fonction récursive échouerait lamentablement avec le problème 67, avec un temps d'exécution démesuré. On le devine, le problème n'est pas tant la multiplication des sous-problèmes à résoudre : il y en a autant que de nombres dans notre triangle, c'est-à-dire raisonnablement peu, que le fait que les sous-problèmes devront être résolus… un très grand nombre de fois. Par exemple, lors de l'invocation de `euler18rec(T, 0, 0)` seront invoqués `euler18rec(T, 1, 0)` et `euler18rec(T, 1, 1)` lesquels invoqueront à leur tour, l'un comme l'autre, `euler18rec(T, 2, 1)` (qui sera donc lui invoqué deux fois).

Pour éviter de refaire plusieurs fois le même travail, on peut faire appel à la mémoïzation : on adjoint à notre fonction récursive un dictionnaire. Lorsque notre fonction est invoquée avec un ensemble de paramètres « nouveaux », on effectue le calcul comme précédemment, mais avant que de renvoyer la valeur calculée, on enregistre dans notre dictionnaire, à la clé formée du tuple des arguments, la valeur obtenue. A contrario, lorsque la valeur demandée a déjà été calculée (ce que l'on voit grâce à notre dictionnaire) il n'y a plus qu'à lire la valeur déjà calculée et renvoyer celle-ci.

Voici comment on pourrait mettre en œuvre ce mécanisme pour le problème 18 :

```
def euler18(T):
    memo = {} # gardera en mémoire toutes les valeurs calculées
    def euler18rec2(i, j):
        # ne pas oublier la condition d'arrêt
        if (i, j) not in memo:
            # le même traitement que pour euler18rec :
            memo[(i, j)] = ...
        return memo[(i, j)]
    return euler18rec2(0, 0)
```

Question 10. Compléter le script fourni pour la fonction `euler18`. Bien sûr, on testera.

Une autre approche s'appuie sur le fait que, comme le montre l'algorithme récursif, la solution optimale depuis un sommet donné est connue dès lors qu'on connaît les solutions optimales depuis les deux sommets qui suivent le sommet initial. L'idée est alors de calculer les solutions optimales à tous les sous-problèmes, en partant du bas ! Ce faisant, on fera manifestement beaucoup plus de travail que par l'algorithme glouton : en pratique autant que de sous-problèmes c'est-à-dire environ $n^2/2$ pour n lignes, mais aussi beaucoup moins que le nombre de chemins possibles (2^{n-1} pour n lignes), lequel correspond à la complexité de notre première solution récursive.

On parle ici de **programmation dynamique** : on résout de proche en proche tous les sous-problèmes en gardant trace de leurs solutions respectives. De manière un peu cachée, la mémoïzation évoquée un peu plus haut rentre aussi dans cette catégorie.

Question 11. Compléter la fonction d'entête `def euler18dynamique(T):` de manière à ce qu'on modifie le triangle `T`, de bas en haut, pour que `T[i][j]` contienne la plus grande somme depuis la position (i, j) jusqu'en bas. (Par exemple, en partant de `T = [[3], [7, 4], [2, 4, 6], [8, 5, 9, 3]]`, on souhaite que `T` soit modifié en `[[23], [20, 19], [10, 13, 15], [8, 5, 9, 3]]`) et il n'y a plus bien sûr qu'à lire et renvoyer `T[0][0]`.

Travail à la maison : les questions 10 et 11 sont de bons sujets de recherche, mais bien sûr, je vous invite à continuer à me soumettre vos propositions pour les exercices de révision distribués au cours du TP précédents. (Il est impératif de savoir faire tous les exercices de niveau 2 et moins.)