TP 10
Algorithmes Gloutons – Sujet

Avant de commencer Avant toute chose, créer un répertoire TP10 dans votre répertoire "Informatique". Sur le site `https://ptsilamartin.github.io/info/TP.html`, télécharger `algorithmes_gloutons.py` que vous copiez dans votre répertoire TP10.

Rappel : On prendra bien soin, dans tout le TP, de documenter les fonctions écrites et de les tester. Les tests seront présentés en commentaires dans le script.

Algorithme Glouton - Remplir un sac à dos

A partir du travail de Nicolas VIDAL et de Nicolas COURRIER pour l'UPSTI.

Un promeneur souhaite transporter dans son sac à dos le fruit de sa cueillette. La cueillette est belle, mais trop lourde pour être entièrement transportée dans le sac à dos. Des choix doivent être faits. Il faut que la masse totale des fruits choisis ne dépasse pas la capacité maximale du sac à dos.

Les fruits cueillis ont des valeurs différentes, et le promeneur souhaite que son chargement soit de la plus grande valeur possible.

Fruits cueillis	Prix au kilo	Quantité ramassée
framboises	24 €/kg	1 kg
myrtilles	16 €/kg	3 kg
fraises	6 €/kg	5 kg
mures	3 €/kg	2 kg

La capacité du sac à dos n'est que de 5 kg.

On suppose que la masse d'un unique fruit est négligeable par rapport à la masse totale du sac en charge.

Présentation de l'algorithme et implémentation

Le principe pour optimiser le chargement est de commencer par mettre dans le sac la quantité maximale de fruits les plus chers par unité de masse. S'il y a encore de la place dans le sac, on continue avec les fruits les plus chers par unité de masse parmi ceux restants, . . . et ainsi de suite jusqu'à ce que le sac soit plein.
Il est possible qu'on ne puisse prendre qu'une fraction de la quantité de fruits disponible, si la capacité maximale du sac est atteinte. L'algorithme sera donc constitué d'une structure itérative, incluant une structure alternative.

Dans `Algorithmes_gloutons.py`, nous avons ébauché l'algorithme de résolution du problème et implémenté `cueillette` par la liste des fruits **triée** par prix au kilo décroissant.
Un fruit est représenté par une liste `fruit=[prix au kilo,nom,quantité ramassée]`.

`cueillette = [[24,"framboises",1], [16,"myrtilles",3], [6,"fraises",5], [3,"mures",2]]`.

Question 1 Dans `Algorithmes_gloutons.py`, compléter la fonction `sac_a_dos(L:list, capacite:int)` qui prend en argument une liste de listes `L` modélisant la cueillette et la capacité du sac à dos, et appliquant la méthode décrite précédemment. Cette fonction renvoie :

- une liste de listes des fruits contenus dans le sac à dos avec leur masse correspondante de façon à avoir le chargement de valeur maximale ;
- la valeur du chargement.

La liste `cueillette` ne doit pas être modifiée. Vous pourrez si nécessaire utiliser une copie de la liste `cueillette` que vous pourrez modifier. La copie de liste de listes se fait avec les instructions :

```
import copy \# au début de votre script\\
L=copy.deepcopy(votreListe) \# quand cela est nécessaire
```

Question 2 A la suite du script `Algorithmes_gloutons.py`, taper les lignes de code permettant d'utiliser la fonction `sac_a_dos( )` pour afficher les fruits et les quantités à choisir pour remplir le sac à dos, à partir de la liste `cueillette`. Exécuter votre programme et vérifier le résultat.

Une variante de ce problème ne trouve pas de solution optimale par la méthode gloutonne. Nous l'étudions ci-après.

Version non fractionnaire du problème du sac à dos

On suppose maintenant que les éléments à transporter ne sont pas fractionnables. Les fruits parmi lesquels choisir sont présentés dans le tableau suivant.

Fruits	Prix au kilo	Masse d'un fruit	Quantité disponible
melon de cavaillon	3 €/kg	1 kg	1
melon jaune	2.5 €/kg	2 kg	1
pastèque	2 €/kg	3 kg	1

Cet ensemble de fruits est modélisé dans `Algorithmes_gloutons.py` par :

- un fruit non fractionnable est représenté par une liste `prix au kilo,nom,masse, quantité disponible`
- une liste de listes `fruitsDisponibles = [[3,"melon de cavaillon",1,1], [2.5,"melon jaune",2,1], [2,"pastèque",3,1]]`.

L'objectif est toujours de placer dans le sac à dos le chargement de valeur maximale, de masse totale inférieure à 5kg. Par contre, les éléments n'étant fractionnables, il est possible que les choix successifs mène à un chargement qui ne remplit pas complètement le sac à dos.

On se propose de tester la méthode gloutonne pour cette nouvelle formulation.

Question 3 Dans `Algorithmes_gloutons.py`, copier-coller `sac_a_dos(L, capacite)` que vous renommerez `sac_a_dos_V2(L, capacite)` et adapter la fonction de sorte qu'elle applique la méthode gloutonne à la version non fractionnaire du problème. Pour cela, il faut : Exécuter la fonction en utilisant la liste `fruitsDisponibles`.

Question 4 Le résultat obtenu est-il optimal ? Comparer avec la solution `S = [2.5, "melon jaune",2,1], [2,"pastèque",3,1]]`. Quelle est la solution dont la valeur est maximale ?

Question 5 De la propriété du choix glouton et de la propriété de la sous-structure optimale, laquelle n'est pas respectée dans cette formulation du problème ? En quoi l'autre propriété l'est ?

Cette version du problème du sac à dos n'est pas adaptée à la méthode gloutonne. Le cours de deuxième année vous permettra d'aborder la programmation dynamique qui offre une solution optimale à ce problème : son principe consiste en l'étude de toutes les combinaisons possibles afin de ne retenir que la combinaison optimale.

Le problème du choix d'activité

Présentation du problème

Le problème du choix d'activité (PCA) concerne le problème d'ordonnancement de plusieurs activités qui rivalisent pour l'utilisation exclusive d'une ressource commune. Une application concrète est l'allocation d'une salle à différents conférenciers au cours d'une journée. Plusieurs personnes souhaitent effectuer une conférence, mais ces conférences ont lieu à des horaires définis imposés par la disponibilité de chaque conférencier.
Certaines conférences démarrent alors que d'autres ne sont pas encore terminées. Il n'est donc pas possible d'attribuer la salle à chaque conférencier pour chacune de ses conférences, et il faut faire des choix.
La solution optimale est celle qui permettra au plus grand nombre de conférences de se tenir.

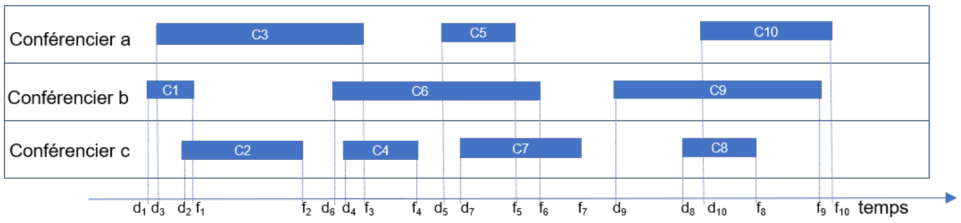

Figure 1 – Conférences parmi lesquelles choisir pour établir le planning de la salle de conférences

Les conférences sont définies par leur heure de début et leur heure de fin. La conférence c_i aura pour heure de début $d(c_i)$ et pour heure de fin $f(c_i)$.

Nous supposerons que les conférences ont été triées dans l'ordre croissant de leurs dates de fin. Si nous travaillons avec un ensemble C de n conférences, on a :

$$f(c_1) < f(c_2) < \ldots < f(c_{n-1}) < f(c_n)$$

On a bien affaire ici à un problème d'optimisation :

- soit $S = \{s_1, s_2, \ldots, s_k\}$ l'ensemble des conférences constituant la solution ;
- l'objectif : maximiser le nombre de conférences : maximiser $|S|$;

- la contrainte : les conférences doivent être compatibles, pour tout $i, j \in \{1, \ldots, k\}$, on doit avoir $f(s_i) \leqslant d(s_j)$ ou $f(s_j) \leqslant d(s_j)$.

La solution gloutonne

Dans notre application, le premier choix glouton consiste à choisir la conférence qui termine le plus tôt possible, préservant ainsi la disponibilité de la salle pour les conférences ultérieures.

Lorsqu'un choix de conférence a été fait, on procède de la même manière pour le choix suivant, mais il y a la contrainte de compatibilité : il faut choisir une conférence compatible avec celles qui ont déjà été choisies : On ne peut choisir qu'une conférence qui débute après la fin de la dernière conférence choisie. Si on note C l'ensemble des conférences, et S l'ensemble des conférences qui constituent notre solution optimale, à l'itération k on a : $S = s_1, s_2, \ldots, s_k$ et $S \subset C$.

Alors il faut choisir la conférence $c_i \in C - S$ telle que $d(c_i) \leq f(s_k)$. L'algorithme construit alors un ensemble S de conférences mutuellement compatibles, c'est-à-dire tel que : $d(s_1) < f(s_1) \leq d(s_2) < f(s_2) \leq \ldots \leq d(s_j) < f(s_j)$.

Algorithme et test à la main

Paramètres : C est la liste des conférences, représentées par leurs dates de début et de fin, et leur intitulé `[d(ci),f(ci),'ci']`.

Résultat : La liste `Planning` de taille maximale de conférences compatibles.

Algorithme :

- la liste `planning` est initialisée avec la première conférence ;
- la variable `fin` prend la valeur de la fin de la première conférence ;
- pour tous les éléments c_i de la liste `C`, on teste si $c_i[0]$>=fin, alors on insère c_i dans le planning et on remplace la valeur de `fin` par $c_i[1]$, sinon, on ne fait rien.

Question 6 Exécuter à la main cette méthode en utilisant l'ensemble de conférences illustré sur la figure 1.

Implémentation

On modélise l'ensemble `C` des conférences par une liste dont chaque élément modélise une conférence. Chaque conférence est modélisée par une liste de la date de début de type `float` avec par exemple 8.45 qui correspond à 8h45, la date de fin de type `float`, et l'intitulé de la conférence de type `str`. Cette liste est ordonnée par ordre croissant de la date de fin des conférences. Ainsi, l'exemple représenté sur la figure 1 pourrait être modélisé par la liste :

```
L=[[8,9,"C1"],[8.45,10.3,"C2"],[8.1,11.3,"C3"],[11.15,11.45,"C4"],[12,12.45,"C5
"],[11,13,"C6"], [12.3,14,"C7"],[16,17,"C8"],[15,18,"C9"],[16.2,18.15,"C10"]]
```

Question 7 Écrire la fonction `choix_conferences(C:list)->list` qui prend en argument une liste de conférences `C`, et qui applique la méthode décrite précédemment. Le résultat est une liste de conférences mutuellement compatibles, de longueur maximale.

Question 8 Écrire les instructions permettant d'utiliser la fonction `choix_conferences(C:list)->list` pour déterminer la liste des conférences retenues parmi les conférences listées dans `L`. Comparer le résultat avec celui obtenu avec le test à la main.

On observe que le résultat obtenu est bien un optimal. Cependant, il existe d'autres solutions, toutes autant optimales, par exemple C2, C4, C7, C10 ou encore C1, C4, C5, C9.

Le choix glouton et le sous problème optimal

Un algorithme glouton ne mène pas forcément à une solution optimale d'un problème d'optimisation. Pour qu'un algorithme glouton soit adapté à la résolution d'un problème, il faut que le problème ait la propriété de sous-structure optimale, c'est-à-dire que la solution optimale du problème contient des solutions optimales des sous-problèmes.

Une résolution par la méthode gloutonne nécessite par ailleurs que les choix gloutons successifs engendrent une solution optimale globalement. C'est la propriété de choix glouton.

Pour notre application : $S = s_1, s_2, \ldots, s_k$ est le résultat de notre algorithme. On montre par récurrence que pour tout $j \leq k$, le sous-ensemble $S_j = s_1, s_2, \ldots, s_j$ de S est un sous ensemble d'une solution optimale :

- Pour j=0 : l'ensemble S_j est un ensemble vide, et est donc un sous-ensemble d'une solution optimale,
- Pour j>0 : Soit T une solution optimale telle que pour $j < k$, $S_j \subset T$ (T contient les activités $s_1, s_2, \ldots, s_j$). On note b l'élément suivant s_j dans T :
  - On a $f(s_{j+1}) \leq f(b)$ (en raison du choix glouton, on choisit l'activité que finit le plus tôt),
  - les éléments de T sont tous compatibles, ainsi chaque élément succédant à b débute après la fin de b et de s_{j+1}.

  Ainsi $T - \{b\} \cup \{s_{j+1}\}$ est une solution optimale, de même cardinal que S, et elle contient $s_1, s_2, \ldots, s_{j+1}$

S est donc une solution optimale. On a bien démontré que l'algorithme du choix d'activité est correct, présente la propriété de sous-structure optimale, et que le choix glouton permet d'arriver à une solution optimale globalement.

Question 9 Écrire la fonction `choix_conferences_recursif(C:list,k:int,n:int)->list` qui prend en argument une liste de conférences `C`, un indice `k` et `n` la taille de C, et qui applique la méthode décrite précédemment de manière récursive. Le résultat est une liste de conférences mutuellement compatibles, de longueur maximale.

Références :

- Algorithmique – Cormen Leiserson Rivest Stein - Dunod
- Matroïdes et algorithmes gloutons : UNE INTRODUCTION - Pierre Béjian
- Algorithmes gloutons - Ressource éduscol NSI
- Algorithmes gloutons - Emmanuel Beffara - lis-lab