ITC – TD n°9

Le problème du sac à dos

On s'intéresse ici à un problème classique d'algorithmique : le problème du sac à dos. La question est la suivante : disposant d'un sac à dos de contenance limitée, et d'un ensemble d'objets ayant un certain poids et une certaine valeur, comment remplir le sac à dos sans dépasser sa contenance et en maximisant la valeur emportée ? Il s'agit d'un problème dit NP complet, c'est-à-dire qu'il n'existe pas d'algorithme de complexité polynomiale trouvant la solution optimale au problème.

L'ensemble des objets sera stocké dans une liste `objets` de tuples (a_i, p_i, v_i) contenant le nom, le poids et la valeur de chaque objet. On peut accéder à une de ces caractéristiques de l'objet i par `objets[i][INOM]`, `objets[i][IPOIDS]` et `objets[i][IVAL]`.

Ici, nous allons nous intéresser à trois approches pour ce problème :

- l'algorithme d'exploration systématique, très coûteux en temps de calcul
- l'algorithme glouton, qui donne très rapidement une solution raisonablement correcte
- l'algorithme en programmation dynamique, qui exploite les spécificités du problème

pour accélérer l'exploration en ne testant pas certaines options inutiles

Sépcification

Toutes ces fonctions `exhaustif`, `glouton` et `dynamique` prendront en paramètre une liste d'objets $[(a_i, p_i, v_i)]$ (de taille N) et un entier c représentant la contenance du sac, et renverront une paire contenant la valeur totale du sous-ensemble d'objets choisis et une liste contenant le sous-ensemble en question.

On manipulera beaucoup de listes de listes ou listes de tuples dans ce TD. La fonction `deepcopy` est importée de façon à faire des copies efficaces et bien séparées des listes originales.

I. Deux fonctions utilitaires

1 – Coder une fonction `calculerValeur` qui prend en paramètre une liste d'objets et calcule la valeur totale de cette liste d'objets.

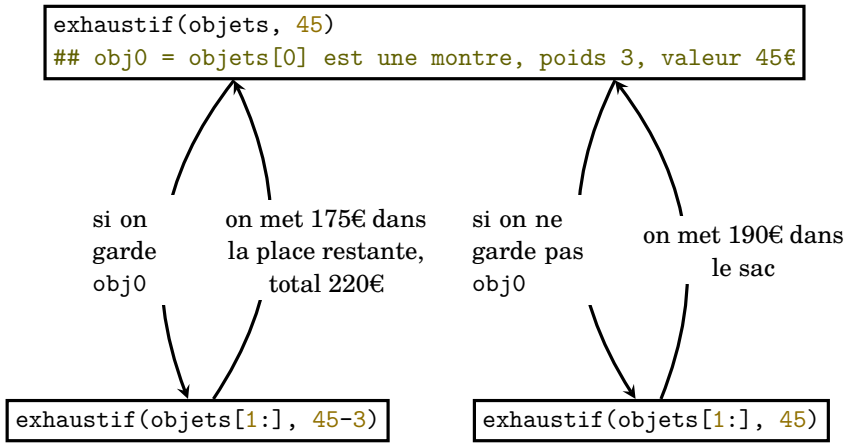
2 – Coder une fonction `calculerPoids` qui prend en paramètre une liste d'objets et calcule le poids total de cette liste d'objets.

II. Exploration exhaustive

On va explorer toutes les configurations possibles à l'aide d'un algorithme récursif testant toutes les possibilités. L'idée (inspirée de la génération exhaustive des permutations d'une liste) est la suivante :

- si la liste d'objets proposés est vide, on renvoie la liste vide avec une valeur de zéro ;
- sinon, on compare la valeur du meilleur remplissage sans le premier objet (mais avec tous les autres objets) avec la valeur du meilleur remplissage avec le premier objet imposé dans le sac (et potentiellement les autres dans la place restante), et on renvoie la meilleure des deux configurations.

Il est à noter que tester une configuration contenant un objet imposé de poids p dans un sac de contenance c revient à tester une configuration dans un sac vide de contenance $c - p$. Cela donne, par exemple, le cas suivant :



renverra un sac à dos où on a choisi de garder l'objet en position 0.

3 – Écrire la fonction `exhaustif` correspondante qui renvoie la solution au problème.

III. Algorithme glouton

Ici, l'algorithme glouton prend à chaque instant l'objet de meilleur rapport valeur/poids qui rentre dans le sac, puis cherche à remplir la place restante. Cet algorithme est donc très rapide (la complexité est limitée par le tri des objets, donc en $\mathcal{O}(N \ln(N))$), mais dans ce problème il n'est pas du tout assuré que l'on obtienne la meilleure solution à la fin. On espère simplement que la solution obtenue soit assez bonne.

4 – Écrire la fonction `glouton` correspondante qui renvoie la solution au problème en triant les objets par rapport valeur/poids. Pour faire rapidement une copie triée de la liste d'objets (remarque : ça n'est pas indispensable, ça dépend de l'approche que vous choisissez), on pourra utiliser les lignes suivantes :

```
l = deepcopy(objs)
l.sort(key=lambda e:e[IVAL]/e[IPOIDS],reverse=True)
```

IV. Programmation dynamique

L'algorithme par programmation dynamique consiste à explorer progressivement l'ensemble des solutions à des problèmes plus simples, et de plus en plus compliqués ; on gagne en efficacité en construisant les solutions aux problèmes compliqués à partir de celles des problèmes simples. L'idée est donc de construire :

- pour un ensemble vide d'objets, les solutions optimales pour un sac de taille 0, puis 1...jusqu'à c ;
- pour un ensemble contenant uniquement le premier objet (a_1, p_1, v_1), les solutions optimales pour un sac de taille 0, puis 1...jusqu'à c ;
- pour un ensemble contenant uniquement les objets $[(a_1, p_1, v_1), (a_2, p_2, v_2)]$, les solutions optimales pour un sac de taille 0, puis 1...jusqu'à c ;
- et ainsi de suite jusqu'à l'ensemble de tous les objets.

On stocke l'ensemble de ces réponses dans deux matrices de taille $(c+1) \times (N+1)$: $M_{i,j}$ contient la valeur maximale stockée dans un sac de taille i à partir des j premiers objets, et $\Gamma_{i,j}$ contient la liste des numéros d'objets à choisir pour obtenir cette valeur. Ces matrices se remplissent par récurrence, d'abord toutes les lignes à colonne j fixée puis on passe à la colonne suivante, comme suit :

- si $i = 0$ ou $j = 0$, $\Gamma_{i,j}$ est la liste vide et $M_{i,j} = 0$
- sinon, si le j^{e}objet est trop grand pour entrer dans le sac de taille i ($i < j$) : on reprend la configuration optimale dans le cas où on n'utilisait pas cet objet j : $\Gamma_{i,j} = \Gamma_{i,j-1}$ et $M_{i,j} = M_{i,j-1}$
- sinon, on compare deux possiblités : soit on prend l'objet j, auquel cas il reste la place $i - p_j$, et on peut lire la configuration optimale dans cet espace restant en $\Gamma_{i-p_j,j-1}$; soit on ne prend pas l'objet j, et la configuration reste celle de $\Gamma_{i,j-1}$; on inscrit alors dans $\Gamma_{i,j}$ la meilleure de ces deux configurations, et $M_{i,j} = \max(M_{i,j-1}, M_{i-p_j,j-1} + v_j)$

La meilleure réponse au problème est donc celle de la case c, N de la matrice $\Gamma_{i,j}$.

Traitons un exemple : pour les objets $(a, 3, 4)$, $(b, 2, 3)$, $(c, 1, 2)$ et un sac de taille $c = 5$, on construira les matrices ci-dessous (on rappelle que les indices démarrent à 0) :

$$M_{i,j} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & \underline{3} & 3 \\ 0 & 4 & \underline{4} & \mathbf{5} \\ 0 & 4 & 4 & 6 \\ 0 & 4 & 7 & 7 \end{pmatrix} \quad \text{et} \quad \Gamma_{i,j} = \begin{pmatrix} \{\} & \{\} & \{\} & \{\} \\ \{\} & \{\} & \{\} & \{c\} \\ \{\} & \{\} & \{\underline{b}\} & \{b\} \\ \{\} & \{a\} & \{\underline{a}\} & \{\mathbf{b}, \mathbf{c}\} \\ \{\} & \{a\} & \{a\} & \{a, c\} \\ \{\} & \{b\} & \{a, b\} & \{a, b\} \end{pmatrix}$$

La case en gras étant obtenue en comparant les cases soulignées, et en ajoutant l'objet c ($j = 3$) à la case de la ligne $i - p_j = 3 - 1 = 2$.

5 – Écrire une fonction `initMatrices` qui prend les tailles du problème et créé, initialise à zéro et renvoie les deux matrices utiles.

6 – Écrire une fonction `comparaisonDynamique(matriceValeurs, i, j, obj)` qui permet de décider s'il vaut mieux ajouter l'objet obj de la colonne j dans un sac de taille i : elle renvoie `True` s'il vaut mieux ajouter l'objet, et `False` sinon.

7 – Écrire la fonction `dynamique` qui renvoie la réponse au problème.