Autour du sac à dos

Introduction

Le problème du sac à dos, noté également KP (knapsack problem), est un problème d'optimisation combinatoire. Il modélise une situation analogue à celle du remplissage d'un sac à dos ne pouvant pas supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale (le profit), avec la contrainte de ne pas dépasser le poids maximum admissible (la quantité totale de ressources disponibles).

D'un point de vue mathématique, la formulation du problème du sac à dos est la suivante :

$$\textit{Maximiser le profit total } P = \sum_{i=0}^{n-1} p_i x_i \textit{ sous la contrainte } \sum_{i=0}^{n-1} r_i x_i \leq b$$

où

- n est le nombre d'objets candidats,
- i est l'entier caractérisant l'objet ($i \in [\![0, n-1]\!]$),
- p_i est le profit (ou valeur) associé à l'objet i,
- x_i est la variable de décision associée à l'objet i : $x_i = 1$ si l'objet i est sélectionné et $x_i = 0$ sinon,
- r_i est la quantité de ressources consommée par l'objet i (ou poids de l'objet i),
- b est la quantité totale de ressources disponibles (ou poids total).

On ne considère que des cas où chaque ressource r_i, chaque profit p_i et la quantité de ressources disponibles b, sont des entiers.

On fait aussi l'hypothèse que $\forall i \in [\![0, n-1]\!], r_i \leq b$.

Dans le problème du sac à dos multidimensionnel, noté MKP, on considère plusieurs sacs à dos ayant chacun un poids maximum admissible. L'objectif est alors de maximiser la valeur totale des objets contenus dans l'ensemble des sacs à dos.

Les applications pratiques sont nombreuses : transport de marchandises, découpe de matériaux, gestion de portefeuilles financiers, allocation de tâches à des systèmes multiprocesseurs, ...

1 Base de données - Exemple de fret maritime de conteneurs

On donne les tables `NAVIRES` et `CONTENEURS` suivantes :

- `NAVIRES(idN,evp,portDepN,dateDep,portDestN)`
  - idN : clé primaire, identifiant du navire (type entier, non nul)
  - evp : capacité en équivalent conteneurs de longueur 20 pieds (type entier)
  - portDepN : port de départ (type chaîne de caractères)
  - dateDep : date de départ (type date)
  - portDestN : port de destination (type chaîne de caractères)

idN	evp	portDepN	dateDep	portDestN
12	1500	"Marseille"	2025-06-23	"Hambourg"
2	16000	"Valence"	2025-06-18	"Algésiras"
5	500	"Le Havre"	2025-10-06	"Rotterdam"
8	23000	"Hongkong"	2025-07-02	"Shanghai"
...	...	...	...	...

- `CONTENEURS(idC,idN,taille,portDepC,dateDisp,portDestC,val)`
  - idC : clé primaire, identifiant du conteneur (type entier)
  - idN : identifiant du navire attribué (type entier, valeur 0 si non encore attribué).
  - taille : longueur du conteneur, 20 ou 40 pieds (type entier). La hauteur et la profondeur sont identiques.
  - portDepC : port de départ (type chaîne de caractères)
  - dateDisp : date de mise à disposition (type date)
  - portDestC : port de destination (type chaîne de caractères)
  - val : tarification, valeur entière de 1 à 5 par ordre croissant de profit pour l'entreprise (type entier)

idC	idN	taille	portDepC	dateDisp	portDestC	val
103	12	20	"Marseille"	2025-08-08	"Hambourg"	2
218	12	40	"Marseille"	2025-07-08	"Valence"	1
5	0	40	"Le Havre"	2025-10-01	"Shangai"	5
885	8	20	"Hongkong"	2025-07-01	"Barcelone"	1
...	...	...	...	...	...	...

Pour les deux questions qui suivent, on demande d'écrire les requêtes en langage `SQL`.

On notera que pour les valeurs d'attributs de type date (format AAAA-MM-JJ), les relations d'ordre ($<, \leq, >, \geq, =, \neq$) sont autorisées. Par exemple, 2025-07-02 < 2025-10-03.

❏ **Q1** – Écrire une requête permettant de retourner la liste des identifiants de conteneurs et leur ratio tarification/longueur trié par ordre décroissant, partant de Marseille et devant aller à Barcelone, avec une mise à disposition avant le 01/01/2025.

❏ **Q2** – Écrire une requête permettant de retourner les identifiants de navire et leur nombre respectif de conteneurs attribués.

2 Structure de données pour l'espace des objets

On demande d'utiliser exclusivement le langage Python pour toute la suite du sujet.

On implémente l'espace des objets i ($i \in [\![0, n-1]\!]$) avec une structure informatique `obj` de type liste de taille n.

La valeur de `obj[i]` est un tuple `(ri,pi)` où `ri` et `pi` sont les valeurs respectives des paramètres r_i et p_i de l'objet i.

Par exemple, `(10,5)` permet de définir pour un objet étiqueté 3 les paramètres $r_3 = 10$ et $p_3 = 5$.

Une solution au problème du sac à dos $S = [x_0, x_1, ..., x_{n-1}]$, est implémentée par une liste Python. On rappelle que $x_i = 1$ si l'objet i est sélectionné et $x_i = 0$ sinon.

Pour les deux questions qui suivent, la fonction python `sum` ne sera pas utilisée.

❏ **Q3** – Écrire une fonction `profit(obj:[(int)],S:[int])->int` prenant en argument une liste `obj` et une liste `S`, et retournant la valeur du profit `P`.

❏ **Q4** – Écrire une fonction `contrainte(obj:[(int)],S:[int],b:int)->bool` prenant en argument une liste `obj`, une liste `S` et un nombre `b`, et retournant le booléen `True` si la contrainte de ressources du problème du sac à dos est respectée, `False` dans le cas contraire.

3 Structure de données pour l'espace des solutions

On choisit de modéliser l'espace des solutions au problème du sac à dos avec un arbre binaire (figure 1). On ne s'intéresse pas pour le moment au respect de la contrainte $\sum_{i=0}^{n-1} r_i x_i \leq b$.

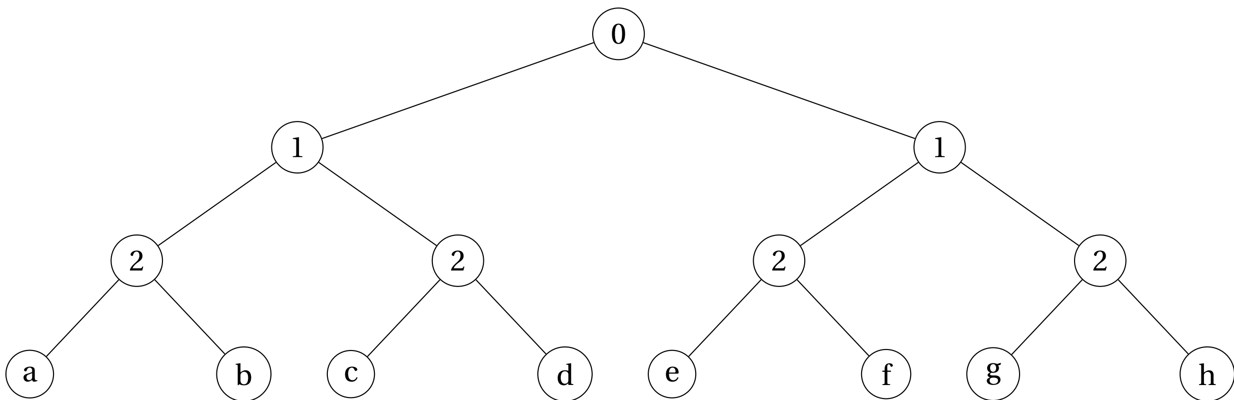

Figure 1 – Arbre binaire et espace des solutions pour le cas particulier où $n = 3$

À chaque nœud, deux choix sont possibles :
— Fils gauche : l'objet i ($i \in [\![0, n-1]\!]$) est sélectionné ($x_i = 1$).
— Fils droit : l'objet i ($i \in [\![0, n-1]\!]$) n'est pas sélectionné ($x_i = 0$).

Au niveau le plus élevé (la racine), le choix se porte sur l'objet 0. Au niveau immédiatement inférieur, le choix se fait sur l'objet 1, et ainsi de suite ... jusqu'à l'objet $n-1$.

Au niveau le plus bas, sont définies les feuilles (sans successeur). À chacune d'entre-elles correspond une solution du type $S = [x_0, x_1, ..., x_{n-1}]$ décrivant les choix effectués le long du chemin menant de la racine à la feuille considérée.

❏ **Q5** – Sur l'exemple de la figure 1 où $n = 3$, à quoi est égale la liste S pour les feuilles b et c ?

❏ **Q6** – Donner le nombre de feuilles de l'arbre binaire en fonction du nombre d'objets n. Indiquer en la justifiant la complexité temporelle en fonction du nombre d'objets n d'un algorithme de résolution du problème du sac à dos de type "force brute" qui envisagerait toutes les combinaisons possibles de sélection d'objets.

Lorsque le nombre d'objets n devient grand, un parcours de toutes les solutions n'est pas possible en un temps raisonnable.

4 Résolution approchée par un algorithme glouton

On envisage une stratégie gloutonne pour la résolution du problème du sac à dos. Lorsque le choix de sélectionner un objet est fait, il ne sera plus remis en question. Les étapes sont les suivantes :

1) On construit la liste `Lqi` qui contient pour chaque objet $i \in [\![0, n-1]\!]$ le rapport profit sur ressource consommée $q_i = p_i / r_i$. En parallèle, on construit la liste `Li`, telle qu'initialement `Li[i]` vaut i. La liste `Li` est telle que `Li[j]` ($j \in [\![0, n-1]\!]$) donne l'indice dans la liste `obj` de l'objet décrit par le rapport situé en position j dans `Lqi`.

2) On trie la liste `Lqi` par ordre décroissant et on modifie simultanément la liste `Li` de sorte que $\forall j \in [\![0, n-2]\!], Lqi[Li[j]] \geq Lqi[Li[j+1]]$.

3) En partant de la quantité totale de ressources disponibles b, on sélectionne l'objet i de rapport q_i le plus grand possible tel que $r_i \leq b$.

4) On met à jour la quantité de ressources disponibles $b = b - r_i$ et on réitère le processus à partir de 3) jusqu'à ce qu'il ne soit plus possible de sélectionner un objet supplémentaire.

❑ **Q7** – On prend le cas particulier où `obj=[(2,3),(1,4),(4,4)]` et `b=5`. Expliquer quelle liste `S` est construite par la stratégie gloutonne précédente.

Le code incomplet de la fonction `constr uitLi(obj:[(int)])->[int]` qui prend en argument la liste `obj` et qui retourne la liste `Li` définie précédemment, est donné ci-après. Cet algorithme correspond aux étapes 1 et 2 décrites en introduction de la stratégie gloutonne.

```
1  def construitLi(obj):
2      Lqi=[]
3      Li=[]
4      for i in range(len(obj)):
5          Lqi.append(.........)
6          Li.append(i)
7      for i in range (1,len(Lqi)):
8          x=Lqi[i]
9          j=i
10         while j>0 and Lqi[j-1]<x:
11             Lqi[j]=Lqi[j-1]
12             Li[j]=.........
13             j-=1
14         Lqi[j]=x
15         Li[j]=.........
16     return Li
```

❑ **Q8** – Proposer le code Python complet des lignes 5, 12 et 15.

❑ **Q9** – Identifier le meilleur des cas et le pire des cas de la méthode de tri utilisée dans la fonction `construitLi` en précisant et justifiant leur complexité temporelle respective.

On donne en page suivante le code incomplet qui calcule la solution au problème du sac à dos `S` sous la forme `S=[x0,x1,...]` avec une stratégie gloutonne.

Une fois la liste `Li` construite en ligne 2, cet algorithme correspond aux étapes 3 et 4 décrites en introduction de la stratégie gloutonne. Les variables `obj` et `b` ont été définies en amont du code.

```
1 S=len(obj)*[0]
2 Li=construitLi(obj)
3 j=0
4 while .........:
5     if .........<=b:
6         S[.........]=.........
7         b=.........
8     j+=1
```

❏ **Q10** – Proposer le code Python complet des lignes 4, 5, 6 et 7.

La complexité temporelle optimale d'une méthode de tri dans le pire des cas en fonction du nombre n de données à trier est quasi linéaire $C(n) = \Theta(nlog(n))$. On peut en déduire que la complexité optimale de la stratégie gloutonne est aussi quasi linéaire.

❏ **Q11** – On reprend le cas particulier où `obj=[(2,3),(1,4),(4,4)]` et `b=5`. Donner la solution optimale pour ce cas particulier. Conclure sur la pertinence d'une approche gloutonne.

5 Résolution exacte par un algorithme de programmation dynamique

On utilise une méthode dite de programmation dynamique pour résoudre le problème du sac à dos à n objets.

On note pour tout $k \in [\![0, n]\!]$ et tout $r \in [\![0, b]\!]$, $P_{max}(k, r)$ le profit maximum réalisable avec les objets d'indice i compris entre 0 et k non inclus pour une quantité maximale de ressources consommées r.

On pose $P_{max}(0, r) = 0$ pour tout $r \in [\![0, b]\!]$, et on considère acquise la formule de récurrence suivante, donnant pour tout $i \in [\![0, n-1]\!]$ et tout $r \in [\![0, b]\!]$, le profit maximum :

$$P_{max}(i+1, r) = max\{P_{max}(i, r), P_{max}(i, r - r_i) + p_i\}$$

où

- p_i est le profit associé à l'objet i ($i \in [\![0, n-1]\!]$),
- r_i est la quantité de ressources consommée par l'objet i ($i \in [\![0, n-1]\!]$),
- r est la quantité maximale de ressources consommées.

On stocke les valeurs de $P_{max}(k, r)$ dans un tableau T à deux dimensions sous forme de liste de listes.

On rappelle que b est la quantité totale de ressources disponibles.

La fonction `KPprogDynamique(obj:[(int)],b:int)->int` donnée en page suivante, implémente un algorithme de programmation dynamique itératif.