OS01

Quentin Beal

January 2026

Table des matières

1 Problème de sac à dos

Le problème du sac à dos consiste à remplir un sac d'une capacité donnée avec un ensemble d'objets ayant chacun un poids et une valeur, de manière à maximiser la valeur totale sans dépasser la capacité du sac.

1.1 PLNE

Le problème peut être modélisé sous forme de Programme Linéaire en Nombres Entiers (PLNE) comme suit :

1.1.1 Données et Paramètres

Soit n le nombre d'objets disponibles[cite : 4]. Pour chaque objet $i \in \{1, \ldots, n\}$, on définit :
— v_i : la valeur de l'objet i[cite : 4].
— w_i : le poids de l'objet i[cite : 4].
— C : la capacité maximale du sac à dos[cite : 4].

1.1.2 Variables de Décision

On définit une variable binaire x_i pour chaque objet i :

$$x_i = \begin{cases} 1 & \text{si l'objet } i \text{ est sélectionné} \\ 0 & \text{sinon} \end{cases}$$

1.1.3 Modèle Linéaire

Le programme s'écrit alors :

$$\text{Maximiser } Z = \sum_{i=1}^{n} v_i x_i \tag{1}$$

Sous les contraintes :

$$\sum_{i=1}^{n} w_i x_i \leq C \quad \text{(Contrainte de capacité)} \tag{2}$$

$$x_i \in \{0, 1\}, \quad \forall i \in \{1, \ldots, n\} \quad \text{(Contrainte d'intégrité)} \tag{3}$$

1.2 Résolution heuristique

1.3 Algorithme glouton

Cet algorithme résout le problème du sac à dos en utilisant une approche gloutonne : il sélectionne les objets dans l'ordre décroissant de leur rapport valeur/poids jusqu'à ce que le sac soit plein

Algorithm 1 Sac à dos glouton (0-1)

```
1:  procedure SacADosGlouton(n, capacit, items, pris)
2:      valeur_totale ← 0
3:      capacit_restante ← capacit
4:      TriParRapport(items, n)                          ▷ Trier par ratio décroissant
5:      for i ← 0 to n − 1 do
6:          pris[i] ← 0                                  ▷ Initialiser à non pris
7:      end for
8:      for i ← 0 to n − 1 do
9:          if items[i].poids ≤ 0 then
10:             continue                                 ▷ Ignorer les poids invalides
11:         end if
12:         if items[i].poids ≤ capacit_restante then
13:             valeur_totale ← valeur_totale + items[i].valeur
14:             capacit_restante ← capacit_restante − items[i].poids
15:             pris[items[i].index_original] ← 1
16:         end if
17:     end for
18:     return valeur_totale
19: end procedure
```

Complexité : $O(n^2 + n) = O(n^2)$ dominée par le tri

1.4 Principe de l'heuristique gloutonne

1. **Calcul du rapport :** Pour chaque objet, calculer $\frac{valeur}{poids}$
2. **Tri :** Ordonner les objets par rapport décroissant
3. **Sélection :** Prendre les objets dans l'ordre tant qu'ils rentrent dans le sac
4. **Arrêt :** S'arrêter quand aucun objet ne rentre plus

Note : Cette approche gloutonne ne garantit pas la solution optimale pour le problème du sac à dos 0-1, mais fournit une heuristique rapide avec une bonne approximation dans de nombreux cas.

1.5 Analyse des résultats Heuristique VS Solveur

instance	Solveur	Glouton
1	855	855
2	840	840
3	806	806
4	3691	3691
5	3942	3944
6	4455	4455

TABLE 1 – Comparaison des résultats entre le solveur optimal et l'algorithme glouton

Sur les instances fournies, nous ne remarquons pas de différences majeures entre les résultats de l'heuristique et ceux du solveur. Notons également que les temps de résolution étaient très courts pour le solveur comme pour l'heuristique (< 1 s). Nous pouvons conclure que nous étions plutôt dans des cas favorables à l'heuristique gloutonne, puisque les différences sont quasiment nulles et que, pour des instances de cette taille, recourir à une méthode heuristique ne semble pas pertinent. En effet, au vu de la rapidité d'execution et de la présence d'une marge d'erreur, même très faible, il apparait plus pertinent de recourir à une méthode exacte.

2 Problème de Bin Packing

Le problème de Bin Packing consiste à ranger un ensemble d'objets, ayant chacun un poids spécifique, dans un nombre minimal de bacs (ou boîtes) de capacité identique. Contrairement au sac à dos, tous les objets doivent être rangés.

2.1 PLNE

2.1.1 Données et Paramètres

— n : le nombre d'objets à ranger.
— C : la capacité maximale de chaque bac.
— w_i : le poids de l'objet $i \in \{1, \ldots, n\}$.
— m : le nombre maximal de bacs disponibles (généralement on prend $m = n$).

2.1.2 Variables de Décision

On définit deux types de variables binaires :
— $y_j = 1$ si le bac j est utilisé, 0 sinon, pour $j \in \{1, \ldots, m\}$.
— $x_{ij} = 1$ si l'objet i est placé dans le bac j, 0 sinon.

2.1.3 Modèle Linéaire

$$\text{Minimiser } Z = \sum_{j=1}^{m} y_j \tag{4}$$

Sous les contraintes suivantes :

$$\sum_{j=1}^{m} x_{ij} = 1, \quad \forall i \in \{1, \ldots, n\} \tag{5}$$

$$\sum_{i=1}^{n} w_i x_{ij} \leq C y_j, \quad \forall j \in \{1, \ldots, m\} \tag{6}$$

$$x_{ij}, y_j \in \{0, 1\}, \quad \forall i, j \tag{7}$$

2.2 Heuristiques

— **First-Fit (FF) :** On place chaque objet dans la première boîte capable de l'accueillir. Si aucune boîte ne convient, on en ouvre une nouvelle.
— **Best-Fit (BF) :** On place l'objet dans la boîte qui laisse le moins d'espace résiduel après insertion (la "meilleure" adaptation).
— **Versions Decreasing (FFD / BFD) :** Les objets sont préalablement triés par poids décroissant, ce qui améliore considérablement les performances.

Algorithm 2 Heuristique First-Fit

```
1: procedure FirstFit(objets, C)
2:     nb_bacs ← 0
3:     capacite_restante ← tableau vide
4:     for chaque objet de objets do
5:         place_trouvee ← faux
6:         for j ← 0 to nb_bacs − 1 do
7:             if capacite_restante[j] ≥ poids(objet) then
8:                 capacite_restante[j] ← capacite_restante[j] − poids(objet)
9:                 place_trouvee ← vrai
10:                break                                  ▷ On s'arrête au premier bac trouvé
11:            end if
12:        end for
13:        if not place_trouvee then
14:            capacite_restante[nb_bacs] ← C − poids(objet)
15:            nb_bacs ← nb_bacs + 1
16:        end if
17:    end forreturn nb_bacs
18: end procedure
```

Algorithm 3 Heuristique Best-Fit

```
1: procedure BestFit(objets, C)
2:     nb_bacs ← 0
3:     for chaque objet de objets do
4:         meilleur_bac ← −1
5:         min_espace ← C + 1
6:         for j de 0 à nb_bacs − 1 do
7:             if espace_libre[j] ≥ poids(objet) et espace_libre[j] − poids(objet) < min_espace then
8:                 meilleur_bac ← j
9:                 min_espace ← espace_libre[j] − poids(objet)
10:            end if
11:        end for
12:        if meilleur_bac ≠ −1 then
13:            Mettre objet dans meilleur_bac
14:        else
15:            Ouvrir nouveau bac, mettre objet dedans, nb_bacs ← nb_bacs + 1
16:        end if
17:    end forreturn nb_bacs
18: end procedure
```

2.3 Analyse des résultats

Instance	FF	FFD	BF	BFD
1	3 bins	3 bins	3 bins	3 bins
2	3 bins	3 bins	3 bins	3 bins
3	3 bins	3 bins	3 bins	3 bins
4	2 bins	2 bins	2 bins	2 bins
5	**3 bins**	**2 bins**	**3 bins**	**2 bins**
6	2 bins	2 bins	2 bins	2 bins

TABLE 2 – Comparaison du nombre de bacs utilisés par chaque heuristique.

Instance	Temps FF (s)	Temps FFD (s)	Temps BF (s)	Temps BFD (s)
1	0.000003	0.000001	0.000003	0.000000
2	0.000004	0.000001	0.000002	0.000002
3	0.000003	0.000002	0.000001	0.000001
4	0.000003	0.000003	0.000002	0.000001
5	0.000006	0.000002	0.000003	0.000002
6	0.000004	0.000003	0.000002	0.000002

TABLE 3 – Temps d'exécution CPU pour les différentes heuristiques.

La borne inférieure $L_1 = \lceil (\sum w_i)/C \rceil$ représente le nombre minimal théorique de bacs requis. Une solution atteignant cette borne est mathématiquement optimale ici nous utilisons la borne inférieure triviale.

Instance	$\sum w_i$	C	Borne L_1	FF / BF	FFD / BFD	Écart (Gap)
Instance 1	501	250	2	3	3	+1
Instance 2	411	205	2	3	3	+1
Instance 3	589	294	2	3	3	+1
Instance 4	2314	1157	2	2	2	0 (Opt)
Instance 5	2404	1202	2	3	2	0 (Opt)
Instance 6	2640	1320	2	2	2	0 (Opt)

TABLE 4 – Comparaison des résultats expérimentaux et de la borne inférieure théorique.

— **Optimalité atteinte :** Les instances 4, 5 (version "Decreasing") et 6 atteignent la borne L_1. Cela démontre que les algorithmes *Decreasing* parviennent à trouver l'optimum global sur ces jeux de données.
— **Impact du tri (Instance 5) :** L'instance 5 illustre parfaitement l'intérêt du tri décroissant : le passage de *First-Fit* à *First-Fit-Decreasing* permet de passer de 3 à 2 bacs, atteignant ainsi la borne inférieure.
— **Limites de la borne L_1 :** Pour les instances 1 et 2, l'écart de +1 persiste. Cela indique souvent que la contrainte d'intégrité des objets (on ne peut pas les couper) rend le remplissage théorique irréalisable en pratique.

Analyse Notons ici que les résultats obtenus via les différentes heuristiques sont très similaires. Les versions "Decreasing" sortent très légèrement du lot puisqu'elles offrent une meilleure solution que leur version classique sur l'instance 5. En terme de Temps de calcul difficile d'évaluer quoi que ce soit sur des instances aussi petites.

3 Partie Modélisation Mathématique

3.1 Données et Paramètres

— c : Nombre d'entités à affecter.
— s : Nombre d'options disponibles.
— v : Nombre de contraintes de volume/capacité.
— Ci_j : Coût unitaire de l'option j, pour $j \in \{1, \ldots, s\}$.
— $T_{k,j}$: Coefficient d'apport de l'option j pour le critère k.

3.2 Variable de Décision

$$X_{i,j} = \begin{cases} 1 & \text{si l'entité } i \text{ est affectée à l'option } j \\ 0 & \text{sinon} \end{cases}$$

3.3 Formulation du Problème

$$\min \quad Z = \sum_{i=1}^{c} \sum_{j=1}^{s} Ci_j \cdot X_{i,j} \tag{8}$$

Sous les contraintes :
— **Unique affectation :** Chaque équipage i doit être affectée à exactement une option.

$$\sum_{j=1}^{s} X_{i,j} = 1, \quad \forall i \in \{1, \ldots, c\} \tag{9}$$

— **Respect des vols :** Chaque vol k, doit être prit au moins une fois.

$$\sum_{i=1}^{c} \sum_{j=1}^{s} X_{i,j} \cdot T_{k,j} \geq 1, \quad \forall k \in \{1, \ldots, v\} \tag{10}$$

— **Domaine des variables :**

$$X_{i,j} \in \{0, 1\}, \quad \forall i \in \{1, \ldots, c\}, \forall j \in \{1, \ldots, s\} \tag{11}$$

3.4 Résultats de l'Optimisation

Suite à la résolution du modèle, les résultats obtenus sont les suivants :
— **Coût total minimal** (Z) : 18
— **Affectations des équipages (Séquences choisies) :**
  — Un équipage affecté à la **Séquence 1** couvre les vols : 1 et 5.
  — Un équipage affecté à la **Séquence 5** couvre les vols : 2, 6, 8 et 9.
  — Un équipage affecté à la **Séquence 12** couvre les vols : 3, 4, 7, 10 et 11.

L'ensemble des vols $\{1, \ldots, 11\}$ est ainsi couvert tout en minimisant les coûts opérationnels, respectant ainsi la contrainte *respectvol*.