PROJET DE SESSION

Apprentissage d'une heuristique à coût réduit par renforcement pour le problème du sac à dos 0-1

Travail effectué par

Antoine Lachance-Loiseau (536 890 354)

Dans le cadre du cours IFT-7020

Travail remis à

M. Claude-Guy Quimper

14 décembre 2025

Table des matières

1 Introduction

Le problème du sac à dos à choix binaire (ou knapsack 0-1 problem) est un problème classique d'optimisation combinatoire qui consiste à sélectionner un sous-ensemble d'objets de façon à maximiser la valeur totale tout en respectant une contrainte de capacité sur le poids. Sur de petites instances du problème, les solveurs modernes sont généralement assez bons et rapides pour trouver la solution optimale. Cependant, pour des instances plus volumineuses, la solution optimale devient rapidement très difficile à trouver puisque le nombre de combinaisons possibles explose. Pour aider les solveurs, on fait appel à des heuristiques simples mais efficaces, comme celle du ratio valeur-poids ($\frac{valeur}{poids}$) des objets. Dans cet article, nous présentons une nouvelle approche pour trouver une solution au problème du sac à dos binaire sous une contrainte de temps donnée.

2 Description du problème

On définit le problème du sac à dos binaire comme suit : on dispose de n objets. Chacun de ces objets a deux attributs : une valeur ($v_i > 0$) et un poids ($p_i > 0$). Le sac à dos, que nous essayons de remplir, a une capacité maximale C.

On cherche donc à maximiser la valeur des objets sélectionnés tout en respectant la capacité maximale du sac à dos.

Pour savoir quels objets sont sélectionnés, on crée un vecteur X de longueur n. Chaque entrée dans $X = (x_1, \ldots, x_n)$ a pour domaine $\{0, 1\}$. Pour chaque position i, la variable $x_i = 1$ indique que l'objet i est sélectionné, tandis que $x_i = 0$ signifie qu'il est exclu.

On cherche :

$$\max \sum_{i=1}^{n} v_i x_i$$

Tout en respectant

$$\sum_{i=1}^{n} p_i x_i \leq C$$

De plus, nous limitons le temps d'exécution du solveur. Cette limite temporelle modifie le but : il ne s'agit plus de garantir une solution optimale, comme dans le problème standard, mais de trouver la meilleure solution avant T millisecondes.

3 Approche proposée

L'approche proposée consiste à apprendre une heuristique de branchement de variables linéaire, par renforcement, combinée à une politique ϵ-greedy. L'idée est d'intégrer cette

heuristique à un solveur pour guider le branchement des variables pendant l'apprentissage et lors de l'utilisation du modèle. Dans cette section, nous allons décortiquer les différents composants de cette approche.

3.1 Solveur Choco

Afin de pouvoir facilement modifier le mécanisme de sélection de variables interne d'un solveur de contraintes, il convient d'utiliser Choco, une bibliothèque à code source ouvert écrite en Java. Choco permet d'implémenter facilement son propre algorithme de branchement grâce aux interfaces qu'il offre. De plus, on peut appliquer une limite de temps d'exécution au solveur, ce qui est essentiel pour la contrainte de temps imposée. En somme, Choco est parfaitement adapté à l'intégration et l'évaluation de l'heuristique proposée.

3.2 Cadre de décision

Afin de bien comprendre les prochaines sections, il convient d'expliquer plus en détails le fonctionnement de l'heuristique ainsi que les éléments qui viennent guider le branchement du solveur. D'abord, pour trouver une solution au problème du sac à dos, le solveur explore un arbre de recherche. À chaque nœud, celui-ci doit brancher une variable : c'est à ce moment que l'heuristique apprise intervient. Celle-ci fonde sa décision sur les caractéristiques clés de deux éléments, soit s l'état actuel du solveur et a la variable candidate. En d'autres mots, a est un objet candidat (donc un objet encore disponible) et s est l'état des autres objets (sélectionnés, exclus ou encore disponibles). Pour extraire les caractéristiques, on fait appel à la fonction $\phi(s, a)$. Celle-ci nous retourne un vecteur de longueur d qui contient des informations clés s et a.

3.3 Caractéristiques (features)

Comme mentionné plus tôt, la fonction $\phi(s, a)$ extrait un vecteur de caractéristiques clés qui décrivent l'état du solveur s et la variable candidate a.

En pratique (voir le fichier *SimpleExtractor.java* et *State.java*), la variable candidate a correspond à un objet du knapsack, caractérisé par sa valeur v_a et son poids p_a. L'état courant du solveur s correspond à une version compacte de l'état réel : on y retrouve la capacité restante du sac c_s, la profondeur actuelle dans l'arbre de recherche d_s, le nombre d'objets total n et le nombre d'objets disponibles n_s.

Voici un tableau qui représente les caractéristiques qui seront pondérées lors de l'apprentissage et utilisées par l'heuristique de branchement.

TABLE 1 – Caractéristiques utilisées pour l'heuristique de branchement

Indice	Définition	Description
$\phi_1(s,a)$	1	Le biais : permet un décalage de la fonction linéaire.
$\phi_2(s,a)$	$\log(1+v_a)$	Le log1p de la valeur de l'objet candidat (le logarithme est utilisé afin de compresser les valeurs élevées).
$\phi_3(s,a)$	$\log(1+p_a)$	Le log1p du poids de l'objet candidat.
$\phi_4(s,a)$	$\log(1+v_a) - \log(1+p_a)$	Approximation logarithmique du ratio $\frac{valeur}{poids}$ utilisé dans l'heuristique gloutonne classique.
$\phi_5(s,a)$	$\log(1+v_a) - \log(1+c_s)$	Approximation logarithmique du ratio $\frac{valeur}{c_s}$.
$\phi_6(s,a)$	$\log(1+p_a) - \log(1+c_s)$	Approximation logarithmique du ratio $\frac{poids}{c_s}$.
$\phi_7(s,a)$	$\log(1+c_s-p_a)$	Capacité restante après l'ajout hypothétique de l'objet.
$\phi_8(s,a)$	$p_a < c_s * 0.1$	Indicateur binaire (0 ou 1) qui indique si l'objet est petit par rapport à la capacité restante.
$\phi_9(s,a)$	$p_a > c_s * 0.5$	Indicateur binaire (0 ou 1) qui indique si l'objet est gros par rapport à la capacité restante.
$\phi_{10}(s,a)$	$\frac{d_s}{n} * \log(1+v_a) - \log(1+p_a)$	Ratio $\frac{valeur}{poids}$ pondéré par la profondeur de recherche
$\phi_{11}(s,a)$	$\frac{n_s}{n} * \log(1+v_a) - \log(1+p_a)$	Ratio $\frac{valeur}{poids}$ pondéré par le ratio d'objets encore disponibles

3.4 Politique ϵ-greedy

L'utilisation de la politique ϵ-greedy permet à l'algorithme d'apprentissage de privilégier, la majorité du temps, l'action estimée la plus prometteuse, tout en explorant occasionnellement des actions de manière aléatoire.

La politique utilise trois hyperparamètres : le taux d'exploration initial ϵ, la dégradation du taux d'exploration ϵ_d et le taux d'exploration minimum $\epsilon_{\min}$. La politique fonctionne comme suit :

D'abord on introduit une variable aléatoire $p \sim \mathcal{U}(0,1)$.

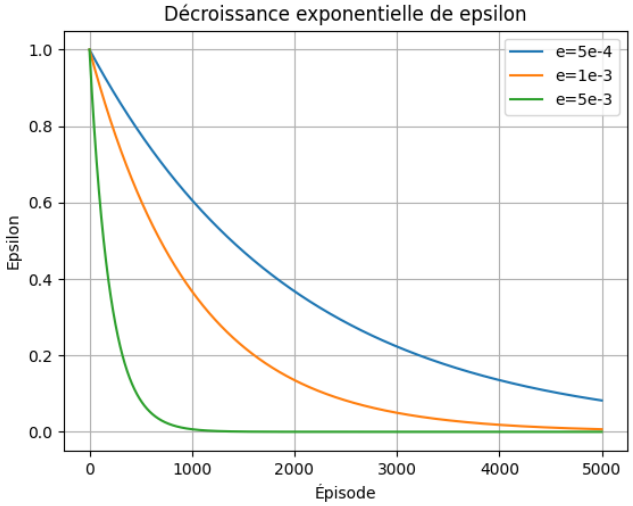
Ensuite on utilise la politique appropriée (exploit ou explore) en fonction de p :

$$\pi(s) \begin{cases} \pi_{\text{explore}}(s), & \text{si } p < \epsilon, \\ \pi_{\text{exploit}}(s), & \text{sinon.} \end{cases}$$

Lors de l'entraînement, par épisodes (donc par instances), on vient mettre à jour la valeur d'ϵ avec une décroissance exponentielle ($RLParams.java$) :

$$\epsilon \leftarrow \epsilon_{\min} + (\epsilon - \epsilon_{\min}) * e^{\epsilon_d}$$

Voici à quoi ressemble la décroissance du taux d'exploration par épisodes pour trois ϵ_d différents :



3.5 L'heuristique proposée

L'heuristique de branchement utilisée repose sur une combinaison linéaire pondérée de plusieurs caractéristiques :

$$f_{\text{score}}(s, a) = w^T \phi(s, a)$$

ou de manière équivalente :

$$f_{\text{score}}(s, a) = \sum_{i=1}^{k} w_i \phi_i(s, a)$$

où k est le nombre de caractéristiques ($k = 11$ pour $SimpleExtractor.java$).

L'heuristique basera donc ses décisions ainsi (voir $RLVariableSelector.java$) :

En exploitation :

$$\pi_{\text{exploit}}(s) = \arg\max_{a \in A(s)} f_{score}(s, a)$$

En exploration (action aléatoire) :

$$\pi_{\text{explore}}(a \mid s) = \frac{1}{|A(s)|}, \quad \forall a \in A(s).$$

où $A(s)$ contient les variables candidates restantes (les objets encore disponibles).

3.6 Apprentissage de l'heuristique

Pour ce qui est de la mise à jour des poids (donc de l'apprentissage), on utilise plusieurs hyperparamètres : le taux d'apprentissage α, le taux d'abandon p_d, la régularisation L2 λ et le taux de mémorisation de la moyenne mobile β. De plus, la fonction de mise à jour dépend d'une récompense R et d'une moyenne mobile b.

D'abord, voici comment R est calculé (voir *RLSolutionMonitor.java*) :

$$\begin{aligned} v_t &= \text{valeur trouvée par le solveur} \\ \Delta_t &= v_t - v_{\text{best}} \\ \Delta_{\max} &\leftarrow \max(\Delta_{\max}, |\Delta_t|) \\ R &= \frac{\Delta_t}{\Delta_{\max}} \\ R &= \max(-1, \min(1, R)) \end{aligned}$$

Donc R est normalisé par $\Delta_{\max}$ et est coupé entre -1 et 1. Cette récompense est calculée lorsqu'une solution a été trouvée par le solveur. Ensuite, on effectue la mise à jour des poids : pour ce faire, on remonte l'arbre de décision et la mise à jour est appliquée avec une probabilité de $1 - p_d$. Donc p_d des nœuds sont ignorés lors de l'apprentissage afin d'aider à diminuer la variance du modèle.

Voici comment les poids sont mis à jour (voir *RLVariableSelector.java*) :

$$w_i \leftarrow w_i + \alpha\Big((R - b)\,\phi_i(s, a) - \lambda\, w_i\Big), \qquad \forall i \in \{1, \ldots, d\},$$

La règle de mise à jour des poids s'inspire de l'algorithme REINFORCE [1]. À noter que la récompense R est appliquée à tous les nœuds également. Donc chaque décision (sélection d'objet) contribue également à la récompense finale. De plus, on utilise b, une moyenne mobile des récompenses, dans le calcul de mise à jour pour diminuer la variance du modèle. b est mise à jour ainsi pour chaque solution trouvée :

$$b \leftarrow b + \beta(R - b)$$

1. `https://cs229.stanford.edu/notes2020fall/notes2020fall/cs229-notes14.pdf`

4 Protocole d'expérimentation

Cette section porte sur la création d'instances complexes du problème du sac à dos, la recherche d'un modèle optimal et les métriques utilisées pour comparer les différentes stratégies.

4.1 Génération d'instances

Pour pouvoir entraîner le modèle décrit antérieurement et pour justifier la présence d'une limite de temps au solveur, il faut des instances du problème assez volumineuses. Pour ce faire, un algorithme Python a été développé pour générer 300 instances avec des paramètres aléatoires.

Voici les paramètres de génération utilisés (voir *knapsack_instance_generator.py*) :

$$\begin{aligned} n_{\text{objets}} &\sim \mathcal{U}(200, 500), \quad n_{\text{objets}} \in \mathbb{N} \\ v_i &\sim \mathcal{U}(1, 10000), \quad v_i \in \mathbb{N} \\ p_i &\sim \mathcal{U}(1, 10000), \quad p_i \in \mathbb{N} \\ c_{\text{ratio}} &\sim \mathcal{U}(0.45, 0.60) \end{aligned}$$

Donc chaque instance a entre 200 et 500 objets, ces objets peuvent avoir une valeur et un poids entre 1 et 10000. c_{ratio} permet de déterminer une capacité maximale en fonction des poids aléatoires. La capacité par instance est calculée comme suit :

$$C = c_{\text{ratio}} \sum_{i=1}^{n_{\text{objets}}} p_i$$

Au total, 300 instances ont été générées de cette manière. Ensuite, le tout a été séparé en deux : un jeu de données d'entraînement de 240 instances et un jeu de données test avec 60 instances.

4.2 Recherche par grille d'hyperparamètres

Pour trouver un modèle performant de manière efficace, il convient de faire une recherche par grille d'hyperparamètres. Voici un tableau des configurations testées :

TABLE 2 – Configurations d'hyperparamètres

ID	α	β	ϵ_d	ϵ	$\epsilon_{\min}$	λ	p_d
1	1e-3	0.05	1e-3	1	0.05	5e-4	0.10
2	8e-4	0.05	8e-4	1	0.03	3e-4	0.05
3	1e-3	0.05	1e-3	1	0.05	5e-4	0.10
4	1.2e-3	0.05	1.5e-3	1	0.07	8e-4	0.15
5	5e-4	0.05	5e-4	1	0.02	2e-4	0.00
6	3e-4	0.05	2e-4	1	0.01	1e-4	0.00
7	7e-4	0.05	5e-4	1	0.03	3e-4	0.05
8	1.5e-3	0.05	2e-3	1	0.08	1e-3	0.20
9	2e-3	0.05	3e-3	1	0.10	5e-4	0.00
10	6e-4	0.05	1e-4	1	0.02	8e-4	0.30

Toutes ces configurations ont été testées sur 100ms et 200ms et avec 3 et 5 epochs d'entraînement. Pour comparer les configurations entre elles, on utilise la moyenne des valeurs trouvées sur les instances. Ensuite, on prend la configuration qui a la plus haute valeur moyenne. Voici la configuration trouvée :

TABLE 3 – Configuration trouvée par la recherche en grille

epochs	timelimit_ms	ID	α	β	ϵ_d	ϵ	$\epsilon_{\min}$	λ	p_d
5	200	4	1.2e-3	0.05	1.2e-3	1	0.07	8e-4	0.15

4.3 Génération des résultats

Afin d'évaluer l'approche proposée, les résultats obtenus sont comparés à ceux des deux techniques de référence : le solveur Choco et son algorithme de branchement par défaut ainsi que Choco combiné avec l'heuristique gloutonne $\frac{valeur}{poids}$. Les 60 instances du jeu de données test ont été résolues par les méthodes avec un temps limite maximal de $T =$ 200ms. Cette limite semble arbitraire, mais elle permet à tous les solveurs de trouver au moins une solution : l'algorithme de branchement par défaut a plus de difficulté avec un délai plus petit.

5 Résultats

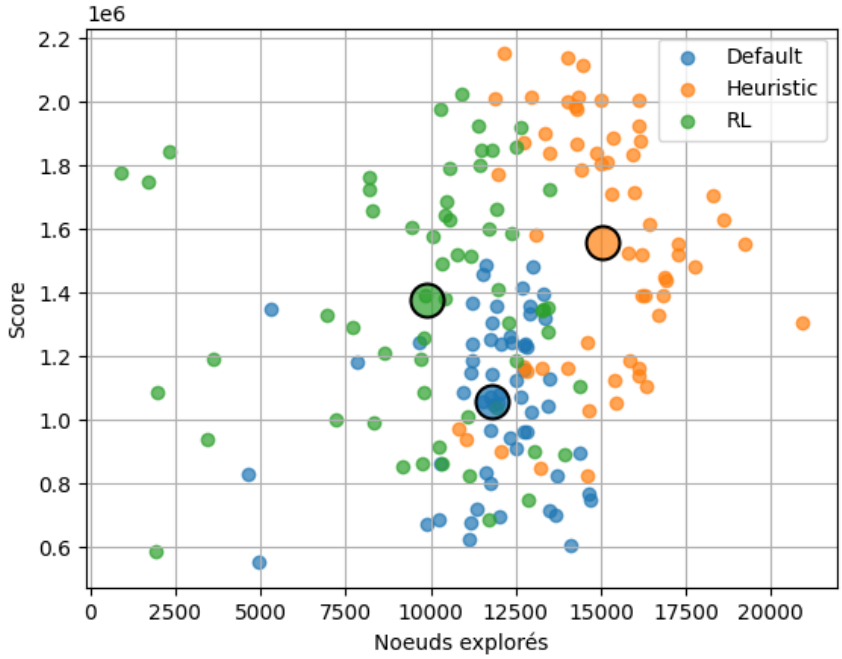
Voici les résultats obtenus pour les trois méthodes sur le jeu de données test de 60 instances avec une limite de temps de $T = 200$ms :

TABLE 4 – Résultats moyens par stratégie avec $T = 200$ms sur les 60 instances test

Stratégie	Score moyen	Nœuds moyens
Solveur	1 057 001.92	11 818.77
Heuristique	1 557 418.25	15 055.52
Approche RL	1 377 336.15	9 908.27

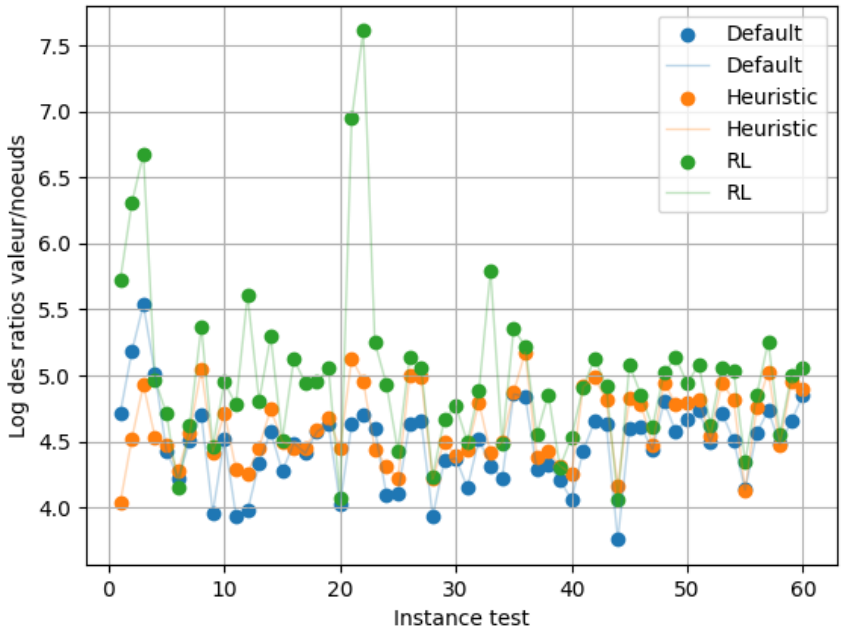
À noter ici que le score moyen représente la valeur maximale du sac à dos que les stratégies ont trouvé pour les 60 instances. Tel qu'on peut remarquer, l'heuristique gloutonne obtient les meilleurs scores moyens. Ce résultat est attendu, car l'heuristique est particulièrement efficace pour trouver de bonnes solutions initiales rapidement. Cependant, on remarque que c'est aussi la stratégie qui a exploré le plus de nœuds : il convient donc de comparer les stratégies en fonction des nœuds qu'ils ont étendus.

Voici un graphique qui montre la valeur maximale obtenue pour le nombre de nœuds étendus par stratégie (la valeur moyenne est encerclée en gros) :



Comme on peut le constater, l'approche proposée obtient de bons résultats en explorant moins de nœuds que les deux autres stratégies. Le modèle proposé est donc plus efficace en termes de qualité par nœuds explorés que les autres techniques. Voici un autre graphique qui montre le ratio $\frac{score}{noeuds}$ qui décrit la qualité par nœuds des différentes approches :

Le graphique montre qu'en moyenne, l'heuristique apprise par renforcement a une meilleure qualité de score par nœuds développés.



6 Discussion

6.1 Retour sur les résultats

Les résultats présentent deux aspects importants de l'approche proposée. D'une part, elle obtient des scores absolus inférieurs à ceux de l'heuristique gloutonne. D'autre part, elle se distingue par une meilleure qualité des solutions par rapport au nombre de nœuds explorés. En pratique, cela signifie que l'heuristique gloutonne reste plus performante en valeur absolue, mais que le modèle par pondération est plus efficace, puisqu'il parvient à des résultats similaires en utilisant moins de ressources.

6.2 Améliorations potentielles

Il existe plusieurs pistes qui pourraient permettre une amélioration des performances de l'approche proposée. D'abord, au niveau du modèle, on pourrait extraire plus de caractéristiques clés afin d'augmenter la «vision» globale de l'approche. De plus, il serait possible de rapprocher le modèle à celui de l'algorithme de REINFORCE en ajoutant une fonction non-linéaire et une descente du gradient pour la mise à jour des poids : cela pourrait augmenter le pouvoir décisionnel du modèle et lui permettre d'apprendre des relations plus complexes entre les caractéristiques.

D'un autre côté, le modèle pourrait bénéficier des résultats de l'heuristique gloutonne en les introduisant à deux endroits. D'abord, le calcul de récompense pourrait être fait sur le meilleur score de l'heuristique gloutonne plutôt que sur le meilleur score trouvé jusqu'à présent par le modèle. Deuxièmement, on pourrait aussi transformer la recherche aléatoire

du ϵ-greedy en une recherche stochastique en pondérant les probabilités de piger un objet par leur ratio $\frac{valeur}{poids}$. Le modèle, en phase d'exploration, aurait plus tendance à prendre des objets de meilleure qualité.

Finalement, l'entraînement du modèle pourrait aussi être plus exhaustif afin de trouver une meilleure configuration que celle utilisée. En d'autres mots, il serait intéressant d'augmenter le nombre d'epochs, le nombre de données d'entraînement ainsi que le temps limite par instance.

7 Conclusion

En conclusion, cet article démontre qu'il est possible d'apprendre une heuristique de branchement pour le problème du sac à dos 0-1, par renforcement, avec une limite temporelle. Bien que l'approche proposée ne surpasse pas l'heuristique gloutonne, elle se démarque avec une meilleure efficacité par nœud exploré, ce qui la rend pertinente dans un contexte où les ressources sont limitées. Les résultats obtenus montrent que cette forme d'apprentissage est prometteuse pour concevoir des heuristiques qui s'adaptent bien aux problèmes d'optimisation combinatoire tels que celui du sac à dos.

Annexe

Code de programmation (en pièces jointes .zip ou sur Github)