Résolution du problème de Sac à dos 0-1

Rapport de projet

Kévin Barreau

8 novembre 2015

Résumé

L'objectif du projet est d'implémenter différentes méthodes de résolution du problème de sac à dos binaire, afin des les comparer. Il s'agit des méthodes du Branch And Bound, de Programmation Dynamique (sous plusieurs implémentations), de Modèle de Flot dans un graphe, ainsi que l'utilisation du solveur Cplex. Le langage de programmation pour atteindre ce but est libre, mais sa rapidité est essentielle pour la résolution. J'ai par conséquent choisi de réaliser le projet en Java, au détriment de C++. Les résultats sont présentés en prenant en compte les comportements des algorithmes suivant les types d'instance (nombre d'objets, capacité maximale du sac, corrélation des objets). L'ensemble du projet peut se retrouver à l'adresse suivante : `https://github.com/LuminusDev/Knapsack/`.

Sommaire

1 Présentation du projet

1.1 Description du problème

Explication du problème de sac à dos, puis sac à dos binaire.

Le problème du sac à dos consiste ranger des objets, possédant chacun un poids et un profit, dans un sac à dos dont la capacité totale est limitée. On cherche alors à maximiser le profit total des objets que l'on met dans le sac à dos, sans que le poids total ne dépasse sa capacité.

Le sac à dos **binaire** est une variante, qui ne permet de prendre un objet que 0 ou 1 fois.

On peut modéliser ce problème par un programme linéaire en nombre entier :

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

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\} \ \forall i$$

Ce problème est **NP-Difficile** au sens faible du terme, car il existe un algorithme le résolvant en un temps pseudo-polynomial (qui dépend de la valeur des données, en l'occurrence la capacité du sac).

1.2 Méthodes de résolution

Pour résoudre ce problème, plusieurs algorithmes ont été implémenté.

— Branch And Bound

— Plus court chemin dans un graphe

— Programme Dynamique (Simple Backward)

— Programme Dynamique (Forward avec liste)

— Programme Dynamique (Core avec liste)

Un modèle du problème est aussi réalisé afin de résoudre les instances dans le solveur CPlex. Cela permet de vérifier les valeurs des solutions des algorithmes implémentés.

Le **Branch And Bound** utilise des bornes supérieures et inférieures pour couper des branches de l'arbre d'énumération des solutions, sur lequel on réalise un parcours en profondeur. Une borne supérieure est donné par la valeur de la solution partielle actuelle dans l'arbre, à laquelle on ajoute la valeur de la relaxation linéaire du problème partielle restant. Une borne inférieure correspond à la valeur de la meilleure solution complète trouvée à l'instant.

Le **programme dynamique en backward** réalise un parcours des solutions aux problèmes partiels du sac à dos binaire, ne gardant que les meilleurs solutions. Grâce à la décomposabilité du problème initial et à une formule de récurrence, on obtient la solution à notre problème à partir des solutions partielles.

Le **programme dynamique en forward** suit le même principe que la méthode en backward, mais utilise des listes pour ne garder les résultats que de états dominants. Cela permet de réduire de manière drastique l'empreinte mémoire du programme.

Le **programme dynamique basé sur le core** est une méthode utilisant le principe de *core* dans un problème de sac à dos, dont l'on augmente la taille pour qu'il couvre l'ensemble du sac à dos, et ainsi obtenir la solution.

La méthode du **plus court chemin dans un graphe** est initialement un problème de flot maximum transformé, afin d'utiliser l'algorithme de Dijkstra. Le graphe est créé selon le même principe que la programmation dynamique.

2 Implémentation des algorithmes de résolution du problème de sac à dos binaire

2.1 Branch And Bound spécialisé

L'algorithme du branch-and-bound implémenté est celui de *Horowitz et Sahni (1974)*, consistant en un parcours en profondeur amélioré sur l'arbre de recherche.

Pour réaliser ce branch-and-bound sur le problème de sac à dos binaire, le choix du branchement (c'est à dire la variable que l'on fixe) se fait sur la mise de l'objet dans le sac, donc 1 si on le prend, 0 sinon. La profondeur de l'arbre de recherche correspond ainsi au nombre d'objets à traiter.

On trie premièrement les objets par ordre décroissant suivant leur ratio profit sur poids. On réalise ensuite des itérations de fixation à 1 pour les objets tant que l'on peut les mettre dans le sac, sinon à 0 en recalculant la borne supérieure, correspondant à la valeur de la solution partielle actuelle à laquelle on ajoute la valeur de la solution du problème linéaire relâché par rapport à la capacité et aux objets restants.

Lorsque l'on arrive sur une feuille de l'arbre de recherche, garde la solution si c'est la meilleure que l'on a trouvé, puis on remonte dans l'arbre jusqu'à une variable fixée à 1 que l'on passe alors à 0, et à partir de laquelle on redémarre une descente.

L'avantage de cet algorithme est qu'il permet d'obtenir une borne inférieure en $\mathcal{O}(n)$, n étant le nombre d'objet. De plus, cette borne correspondant à la solution d'un algorithme glouton consistant à prendre dans l'ordre les objets ayant le meilleur ratio profit sur poids s'il rentre dans le sac. Cette borne, suivant les instances de problème, permet de couper rapidement des branches de l'arbre de recherche, en adéquation avec la borne supérieure.

2.2 Plus court chemin dans un graphe

Il est possible de reformuler le problème du sac à dos binaire comme un problème de plus court chemin dans un graphe. Le graphe est créé suivant les principes de couches, de transitions et d'états. On notera $maxProfit$ la valeur du plus grand profit dans l'ensemble d'objet de l'instance du problème.

Pour notre problème, un **état** est un noeud du graphe, et il représente le profit maximal pour un ensemble d'objets (tous les objets de la couche actuelle et des couches précédentes) et la capacité utilisée.

Les **couches** représentent chacune la prise de décision sur un objet i. Une couche compte au maximum $W + 1$ noeuds (au pire des cas, car on ne crée pas les noeuds qui ne correspondent pas à une solution partielle). Il y a au plus n couches dans le graphe, correspondant au nombre d'objet dans le problème.

Une **transition** correspond à un arc dans le graphe, et relie les couches adjacentes. Chaque noeud possède au maximum 2 transitions sortantes. La première transition, vers l'état de la couche suivante avec la même capacité et un poids de $maxProfit$ sur l'arc, représente le fait de ne pas prendre l'objet de la couche actuelle. la deuxième transition, vers l'état de la couche suivante avec la même capacité + $poids_i$ et un poids de $maxProfit - profit_i$ sur l'arc, représente au contraire le fait de prendre l'objet.

L'utilisation de $maxProfit$ dans les poids des arcs permet de n'utiliser que des valeurs positives ou nulles, permettant de ce fait d'utiliser l'algorithme de Dijkstra sur ce graphe. En ajoutant un dernier noeud faisant office de puit, dont les noeuds de la dernière couche possèdent une transition vers lui (de poids $maxProfit$), on peut réaliser un plus court chemin entre le noeud de l'état de couche 1 avec une capacité de 0, et le puit. Le plus court chemin obtenu permet alors de trouver la solution au problème.

Attention cependant, car pour obtenir la valeur de la solution, il faut prendre en compte le fait d'avoir ajouter $maxProfit$ au poids des arcs, et donc les retirer pour avoir la véritable valeur.

2.3 Programme dynamique (simple backward)

L'implémentation naïve du problème de sac à dos binaire avec un programme dynamique consiste à reprendre les états et les couches de la méthode de résolution dans un graphe. On obtient ainsi une matrice en $\mathcal{O}(nW)$ en mémoire.

Une cellule de cette matrice correspond à $V^k(b)$ qui est le profit maximum qu'on peut obtenir avec les objets 1 à k et une capacité de b, telle que :

$$V^k(b) = \max\{\sum_{i=1}^{k} p_i x_i : \sum_{i=1}^{k} w_i x_i \leq b, x_i \in \{0,1\} i = 1, \ldots, k\}$$

Il suffit ensuite d'avoir une formule de récurrence permettant de calculer les $V^k(b)$ de proche en proche, avec

$$V^k(b) = \max\{\underbrace{V^{k-1}(b)}_{x_k=0}, \underbrace{V^{k-1}(b - w_k) + p_k}_{x_k=1}\}, \text{ pour } k = 1, \ldots, n \text{ et } b = 1, \ldots, W$$

et initialisée telle que $V^0(b) = 0$ pour $b = 0, \ldots, W$.

La valeur de la solution est donnée par $\max_{b \leq W} V^n(b)$.

L'algorithme pour résoudre ce programme dynamique est le suivant :

pour $b = 0$ *à* W **faire**
  $V^0(b) = 0$;
fin
pour $k = 0$ *à* $n-1$ **faire**
  pour $b = W$ *à* w_k **faire**
    si $V^k(b - w_k) + p_k > V^k(b)$ **alors**
      $V^{k+1}(b) = \max\{V^k(b), V^k(b - w_k) + p_k\}$;
    sinon
      $V^{k+1}(b) = V^k(b)$;
    fin
  fin
  pour $b = 0$ *à* $\min\{w_k - 1, W\}$ **faire**
    $V^{k+1}(b) = V^k(b)$;
  fin
fin
retourner $V[W]$

On peut, à partir de là, retrouver la solution en parcourant les objets dans le sens inverse de l'algorithme, avec :

$$X^k(b) = \begin{cases} 1 \text{ si } V^{k-1}(b - w_k) + p_k > V^{k-1}(b) \\ 0 \text{ sinon} \end{cases}$$

On obtient ainsi une algorithme qui résout le problème de sac à dos binaire avec une complexité pseudo-polynomiale (qui dépend de la taille de W) de $\mathcal{O}(nW)$ en temps de calcul et un espace mémoire de $\mathcal{O}(nW)$.

2.4 Programme dynamique (forward en liste)

La méthode du programme dynamique avec une matrice possède un énorme désavantage : son utilisation mémoire. Cela l'empêche d'être utilisé dans un environnement contraint (système embarqué, IoT, ...) ou pour résoudre de très grosses instances. C'est ainsi que l'on se dirige vers un **programme dynamique en liste**.

On ne conserve que le poids et le profit dans un état, ce qui correspond à une solution partielle au problème. L'objectif de la liste est de ne garder que les états utiles au cours de la récursion. On peut le faire grâce au principe de dominance des solutions partielles.

Ainsi, pour un état (w, p) avec w comme poids actuel et p comme profit actuel, on peut appliquer la **dominance** par rapport à un autre état qui indique que :

$$(w^1, p^1) \succ (w^2, p^2), \text{ si } (w^1 < w^2 \wedge p^1 \geq p^2) \vee (w^1 = w^2 \wedge p^1 > p^2)$$

On peut aussi couper les états par borne, dans le cas où $p + PL(n-k, W-w) \leq LB$, avec $PL(a, b)$ la relaxation linéaire du problème pour les objets de $n-k$ à k et une capacité de $W - w$, et LB une borne inférieure de l'instance calculée préalablement à partir d'un algorithme glouton (combinaison du glouton sur les ratios et du glouton sur les profits).

L'algorithme de résolution par la méthode de programme dynamique en liste peut alors se résumer à :

Soit $list = \{(0, 0)\}$
pour $k = 1$ *à* n **faire**
  Soit $list'$ vide
  pour $s = (w, p) \in list$ **faire**
    si $w + w_k \leq W$ **alors**
      Ajouter $(w + w_k, p + p_k)$ à $list'$
    fin
    Fusionner $list'$ et $list$ en éliminant les états dominés
  fin
fin
retourner $max\{p : (w, p) \in list\}$

On ne garde alors qu'un nombre limité d'états dans la liste, qui est inférieur au $\mathcal{O}(nW)$ de la matrice. Pour retrouver la solution, on ajoute un pointeur dans chaque état vers son état précédent. Il est a noté que l'utilisation de pointeur peut augmenter la mémoire nécessaire au fonctionnement de l'algorithme si elle est mal gérée. En effet, garder un pointeur dans un état actif veut dire qu'il ne faut pas supprimer les états précédents ce pointeur. On ne peut donc rien supprimer dans un langage comme C++ où la gestion de la mémoire dynamique doit être réalisée à la main. Dans le cas de ce projet, fait en Java, le GC (garbage collecter) s'occupe seul de supprimer les états n'ayant plus aucun pointeur d'actif, ce qui limite l'empreinte mémoire.

2.5 Programme dynamique (core en liste)

3 Résultats et analyses

Les instances sont créées à l'aide de deux générateurs :

- un générateur pseudo-aléatoire possédant comme paramètres le poids min et max d'un objet, ainsi que le profit min et max d'un objet,
- un générateur linéaire, avec un profit proportionnel au poids. Il permet de créer des instances fortement corrélées (tous avec le même ratio profit/poids par exemple).

Dans les résultats des jeux de données, tous les temps sont exprimés en millisecondes. De plus :

"OoM" : "Out Of Memory" correspondant à une trop grande utilisation de la mémoire ($>$ 2Go).

"-" : temps d'exécution supérieur à 10 minutes.

3.1 Données faiblement corrélées

Le premier jeu de données est constitué d'instances générées aléatoirement, avec un poids inférieur à 25% de la capacité, et un profit inférieur à 50% de la capacité, sur 10 itérations.

A partir de 160K items, le core ne donne pas forcément la bonne solution (dû à l'utilisation du type "int" qui fausse le résultat de la relaxation linéaire sur de trop grand nombre).

	Bab			Backward			Forward			Core			Graphe		
(nbItem, cap)	mean	min	max	mean	min	max	mean	min	max	mean	min	max	mean	min	max
(200,400)	0	0	0	1	0	9	5	0	53	0	0	4	15767	13476	17309
(400,800)	0	0	1	4	0	27	7	0	58	1	0	7	254781	194129	279740
(800,1600)	0	0	1	9	0	19	6	0	48	2	0	9	-	-	-
(1600,3200)	0	0	0	14	0	47	15	0	62	0	0	0	-	-	-
(16K,32K)	0	0	1	OoM	OoM	OoM	50	15	110	19	0	62	OoM	OoM	OoM
(160K,320K)	18	15	47	OoM	OoM	OoM	266	109	594	128	62	289	OoM	OoM	OoM
(320K,640K)	36	5	54	OoM	OoM	OoM	503	227	1414	349	118	612	OoM	OoM	OoM
(640K,1280K)	101	26	166	OoM	OoM	OoM	994	613	1696	1527	257	2826	OoM	OoM	OoM
(1280K,2560K)	142	44	321	OoM	OoM	OoM	1030	630	2462	2098	469	5542	OoM	OoM	OoM

TABLE 1 – Résultats des données faiblement corrélées

On remarque que la méthode de **graphe** est extrêmement longue même avec de petites instances, avec 15 secondes en moyenne pour 200 objets et 400 de capacité, alors que tous les autres algorithmes donnent un résultat quasi instantané jusqu'à 1600 objets et 3200 de capacité. On peut l'expliquer par le fait que cette méthode ne permet pas de couper un ensemble de solution ou d'en éliminer par dominance.

De plus, les méthodes de **backward** et de **graphe** ne peuvent plus donner de résultat à partir de 16K objets à cause d'un dépassement de mémoire. La taille de la matrice de taille $N * W$ en backward et la taille du graphe (du même ordre de grandeur) limite la possibilité d'utilisation de ces algorithmes. Les temps en backward sont cependant bons lorsque la mémoire est suffisante, avec des mesures similaires aux autres programmes dynamiques (forward et core).

Les données faiblement corrélées permettent d'obtenir d'excellentes bornes et coupes par dominance. Ce qui explique des résultats toujours très bons pour le **branch-and-bound**, le **forward** et le **core**. On peut ainsi traiter des instances de très grandes tailles sans subir de réelle contreperformance (pour plus de 1M d'objets, on ne dépasse pas les 5 secondes dans le pire cas, et même 0.321 seconde pour le branch-and-bound).

3.2 Données très fortement corrélées

Le deuxième jeu de données est constitué d'instances générées de manière linéaire, avec un poids égal au profit (ex : (1,1)(2,2)(3,3)...), sur 10 itérations. Les ratios profit/poids sont ainsi identiques.

	Bab			Backward			Forward			Core			Graphe		
(nbItem, cap)	mean	min	max	mean	min	max	mean	min	max	mean	min	max	mean	min	max
(32,64)	0	0	1	0	0	2	6	0	51	3944	3874	4088	24	10	130
(33,66)	0	0	0	0	0	2	13	2	55	6076	5898	6180	40	13	137
(40,80)	0	0	0	0	0	2	36	13	55	81245	83478	84600	219	184	256
(50,100)	0	0	0	0	0	2	6	1	52	-	-	-	52	25	195
(10K,20K)	1	0	4	664	454	968	-	-	-	-	-	-	OoM	OoM	OoM
(100K,200K)	4	2	12	OoM	OoM	OoM	-	-	-	-	-	-	OoM	OoM	OoM
(10M,20M)	391	189	1940	OoM	OoM	OoM	-	-	-	-	-	-	OoM	OoM	OoM

TABLE 2 – Résultats

De la même manière que pour les instances non corrélées, les méthodes de graphe et de backward sont limités par leur utilisation excessive de la mémoire, ce qui ne permet pas de dépasser 10K objets et 100K objets respectivement.

On observe immédiatement un comportement de la méthode du **core** différent des autres méthodes, avec des temps dépassant les 3 secondes en moyenne, pour seulement 32 objets. En ajoutant seulement 1 objet, on augmente le temps moyen de 50%, alors que les autres algorithmes ne donnent pas de différence significatives. On pourrait expliquer ce comportement par le fait que la borne inférieure utilisée dans la méthode est

légèrement inférieure à la solution optimale. L'algorithme réalise ensuite une recherche exhaustive des solutions, ne parvenant pas à supprimer des solutions que ce soit par borne ou par dominance. La méthode du **forward** n'arrive plus à suivre à partir d'un certain seuil de la même manière.

Le **branch-and-bound** permet de trouver des solutions très rapidement, car la première descente donne une borne qui est la solution optimale. L'algorithme résout ainsi ces instances en $\mathcal{O}(n^2)$, d'où des temps inférieur à la seconde pour des instances énormes (10M d'objets par exemple).

3.3 Données fortement corrélées

Le troisième jeu de données est constitué d'instances générées de manière linéaire, avec $poids(i) = minPoids + i$ et $profit(i) = minProfit + i * 0.2$, sur 10 itérations.

	Bab			Backward			Forward			Core			Graphe		
(nbItem, cap)	mean	min	max	mean	min	max	mean	min	max	mean	min	max	mean	min	max
(100,500)	0	0	0	1	0	5	6	0	53	34	25	73	5855	4468	6069
(200,1000)	0	0	0	2	0	11	5	0	53	5	2	18	74278	71470	76133
(1K,5K)	0	0	1	15	6	36	8	2	60	-	-	-	-	-	-
(10K,50K)	0	0	2	OoM	OoM	OoM	27	7	84	-	-	-	OoM	OoM	OoM
(100K,500K)	2	2	8	OoM	OoM	OoM	130	65	370	-	-	-	OoM	OoM	OoM
(220K,1100K)	35	20	80	OoM	OoM	OoM	-	-	-	-	-	-	OoM	OoM	OoM
(2M,10M)	281	149	697	OoM	OoM	OoM	-	-	-	-	-	-	OoM	OoM	OoM
(5M,25M)	390	354	531	OoM	OoM	OoM	-	-	-	-	-	-	OoM	OoM	OoM

TABLE 3 – Résultats

Les résultats de ce jeu de données ressemblent fortement au jeu de données précédent. On remarque tout de même une amélioration des résultats pour les algorithmes de programmation dynamique, avec la méthode du **core** donnant des résultats sous les 10 minutes pour les instances inférieures à 200 objets, contre seulement 50 pour les données précédentes. La méthode de **forward** peut gérer des instances de 100K objets aisément sur ces instances, parvenant donc à couper par des bornes ou de la dominance, là où elle n'y parvenait pour le jeu de données précédent.

3.4 Données fortement corrélées sur le poids uniquement

Le quatrième jeu de données est constitué d'instances générées aléatoirement, avec $poids \in [nbItems/10, nbItems/10 + nbItems/100]$ et $profit \in [1, nbItems]$, sur 10 itérations.

	Bab			Backward			Forward			Core			Graphe		
(nbItem, cap)	mean	min	max	mean	min	max	mean	min	max	mean	min	max	mean	min	max
(200,1000)	6	0	52	3	0	16	6	0	49	1	0	5	70896	56038	73046
(400,2000)	3095	0	20178	4	1	16	23	1	82	8	0	44	-	-	-
(800,4000)	4280	0	31651	10	5	35	37	2	123	16	3	71	-	-	-
(1600,8000)	-	0	-	41	21	83	1284	1	2857	745	1	3432	-	-	-
(10K,50K)	-	-	-	OoM	OoM	OoM	19035	7712	31749	170000	170000	170000	-	-	-

TABLE 4 – Résultats

La particularité de ces résultats par rapport aux autres est de posséder une forte dispersion. On retrouve des instances simples et compliquées au sein d'une même distribution de données.

On note une difficulté pour la méthode de **branch-and-bound** à donner des résultats rapidement, ce qui n'était pas dans les précédents jeux de données. L'augmentation du nombre d'objets influe grandement sur les temps de résolution, avec plus de 10 minutes en moyenne pour résoudre des instances de 1600 objets, alors que plus d'un million d'objets pouvaient être gérés dans les autres jeux de données.

Les méthodes de programmation dynamique permettent cependant d'obtenir de bons résultats sur ces instances, en réussissant à ne pas dépasser les 10 minutes dans le pire cas pour des instances de 10000 objets. Les temps sont tout de même relativement plus long dans la résolution que ce que l'on peut obtenir au mieux dans les jeux de données précédent.

4 Conclusion

En prenant en compte les résultats obtenus sur les jeux de données précédent, on peut remarquer que deux méthodes se démarquent dans la résolution du problème de sac à dos binaire.

En étudiant au préalable la corrélation des données, on serait amener à utiliser la méthode de **branch-and-bound** lorsque les données sont faiblement corrélées ou au contraire très fortement corrélées linéairement. On peut résoudre des instances contenant des millions d'objets de cette manière. A partir du moment où les données deviennent fortement corrélées (de manière non linéaire), cette méthode peut rapidement être dépassée, et dans ce cas on lui préférera la méthode de programmation dynamique, et plus particulièrement la méthode de **programmation dynamique en liste**.