Chapitre 17 : Complément d'algorithmique

Table des matières

1 Optimisation

1.1 Optimisation exacte

1.1.1 Introduction

On s'intéresse ici à la résolution de problèmes d'optimisation au sens de la définition du chapitre 16, en 1.2.2 : on cherche un algorithme calculant une solution optimale **pour toute instance**.

1.1.2 Exemple : le problème du sac à dos

Le problème est le suivant : on dispose d'objets de poids respectifs $w_0, \dots, w_{n-1}$ et de valeurs respectives $p_0, \dots, p_{n-1}$ et d'un sac à dos capable de supporter un poids W. On souhaite sélectionner des objets de sorte à maximiser la valeur totale sans dépasser la capacité du sac à dos.

Dans la version en variables réelles, on suppose que l'on peut prendre des fractions des objets. Le problème d'optimisation se formule ainsi :

Maximiser $\sum_{i=0}^{n-1} x_i p_i$ sous les contraintes :

$$\begin{cases} \sum_{i=0}^{n-1} x_i w_i \leqslant W \\ \forall i \in [\![0\ ;\ n-1]\!]\,,\ x_i \in [0\ ;\ 1] \end{cases}$$

Ce problème est résolu par un algorithme glouton :

Algorithm 1: Solution du problème du sac à dos, version en variables réelles

```
Trier les objets par p_i/w_i décroissant;
while que possible en considérant les objets dans cet ordre do
    Fixer x_i à 1;
Lorsque cela n'est plus possible, prendre la fraction de l'objet courant
  permettant de remplir le sac;
```

Cet algorithme calcule bien une solution optimale : si on note i l'objet de $\frac{p_i}{w_i}$ maximal non encore sélectionné et si une solution optimale coïncidant avec l'algorithme sur les objets avant i, et ne sélectionne pas cet objet dans son intégralité, alors $\exists j$ tel que la solution optimale sélectionne une fraction de l'objet j qui est $> x_j$.

Dans ce cas, il existe $\delta_j > 0$ tel que l'on peut ajouter une quantité $\frac{\delta_j}{w_i}$ de l'objet i et retirer une quantité $\frac{\delta_j}{w_j}$ de l'objet j à la solution optimale.

La variation de poids est $\frac{\delta_j}{w_i} w_i - \frac{\delta_j}{w_j} w_j = 0$, donc on a toujours une solution.

La variation de valeur est

$$\frac{\delta_j}{w_i}p_i - \frac{\delta_j}{w_j}p_j = \underbrace{\delta_j}_{>0}\underbrace{\left(\frac{p_i}{w_i} - \frac{p_j}{w_j}\right)}_{\geqslant 0}$$

Donc la solution reste optimale.

On peut donc modifier la solution optimale jusqu'à l'obtention d'une solution optimale ayant choisi l'objet i dans son intégralité.

L'invariant « il existe une solution optimale ayant fait les mêmes choix que l'algorithme glouton » est vrai.

Remarque : le problème du sac à dos, dans sa version entière (les $x_i \in \{0, 1\}$) ne peut pas être résolu par l'algorithme glouton (algorithme n°1) auquel on retire la dernière étape ne prenant qu'une fraction du dernier objet.

Poids	5	5	7
Valeur	5	5	8

$W = 10$

Cf l'exemple ci-dessus : l'algorithme glouton donne une solution de valeur 8 en prenant l'objet de poids 7 alors qu'une solution optimale est de valeur 10 : on prend les deux objets de poids 5.

1.1.3 Séparation et évaluation (*branch and bound*)

• Une technique de résolution des problèmes d'optimisation consiste à effectuer une exploration exhaustive de l'ensemble des solutions et à conserver la meilleure solution.

Cependant, on se heurte à des problèmes de complexité (exemple : pour le sac à dos en variables entières, il y a 2^n solutions potentielles à tester).

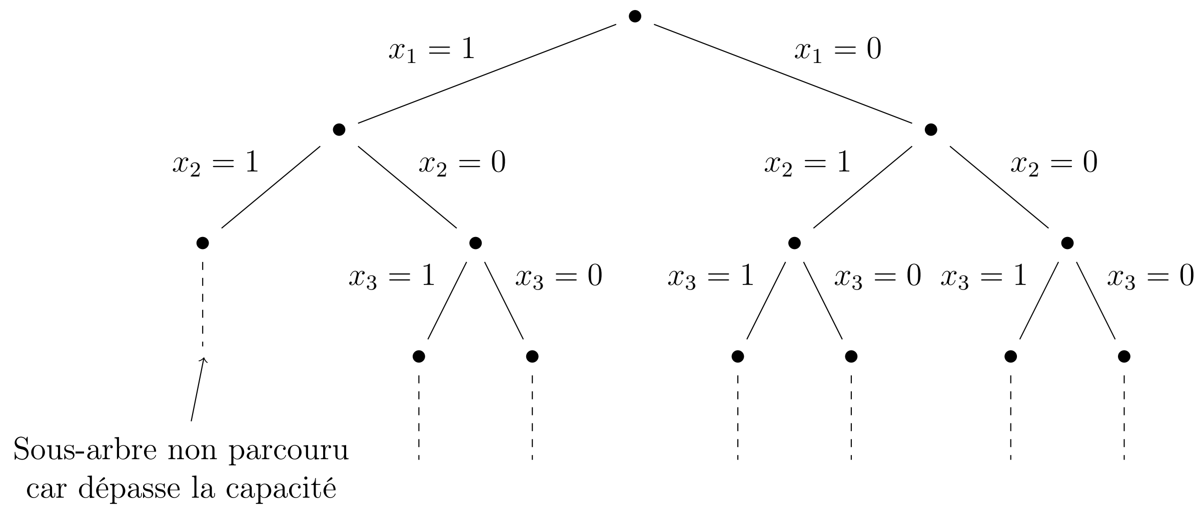
On peut parfois accélérer la recherche grâce à l'heuristique du retour sur trace (*cf* chapitre 8, 4.3).

Par exemple, pour le problème du sac à dos, il y a surement de nombreuses combinaisons d'objets qui dépassent la capacité du sac à dos. On peut donc sélectionner les objets un à un et lorsque l'on s'aperçoit que la capacité du sac à dos est dépassée, on revient sur le dernier choix.

En pratique, cela revient à construire un arbre binaire dans lequel tous les nœuds de même profondeur correspondent à un même objet et pour ces nœuds, le fils gauche correspond au cas où l'on a sélectionné l'objet et le fils droit au cas où l'objet n'est pas sélectionné.

On élague les branches correspondant à des sélections dépassant la capacité du sac à dos.

Si $w_1 + w_2 > W$,



Dans le cadre de la résolution d'un problème d'optimisation, on peut parfois élaguer encore plus l'arbre de recherche en considérant le coût des solutions construites : si on sait évaluer une borne du meilleur possible pour une série de choix sans parcourir l'intégralité du sous-arbre correspondant, on peut parfois élaguer ce sous-arbre si on connaît déjà une solution de coût meilleur que cette borne.

Cette méthode consiste à concevoir un algorithme par séparation et évaluation :

– La séparation consiste à diviser le problème en sous-problèmes, donc à créer un branchement dans l'arbre de reverche.

  Exemple : pour le problème du sac à dos, on a deux sous-problèmes, selon que l'objet i est sélectionné ou non.

– L'évaluation consiste à déterminer une borne sur le coût d'une solution optimale **réalisable avec les choix déjà faits** et à le comparer avec une borne connue pour savoir s'il est nécessaire de poursuivre l'exploration du sous-arbre.

Pour que cette méthode soit efficace, on a besoin de bonnes heuristiques pour :

– La séparation : si les choix initiaux convergent rapidement vers une « bonne solution », on élaguera plus de branches dans la suite de l'exploration.

– L'évaluation : on doit pouvoir calculer *efficacement* une borne *la plus juste possible* pour avoir de bonnes chances d'élaguer des branches.

1.1.4 Exemple : le problème du sac à dos

• Pour la séparation, on sélectionne ou pas un objet, avec l'heuristique suivante : on considère les objets par $\dfrac{p_i}{w_i}$ décroissant.

• Pour l'évaluation, on utilise l'heuristique de relaxation : on relâche certaines contraintes, ce qui élargit le domaine des solutions donc permet potentiellement d'atteindre un meilleur coût. Si le problème relâché est plus simple à résoudre, le coût d'une solution optimale est donc la borne recherchée.

Ici, on effectue une relaxation continue : on n'impose plus aux x_i d'être des entiers, ce qui nous ramène au problème vu en 1.1.2 (page 2), que l'on sait résoudre efficacement (en $\mathcal{O}(n)$ car les objets seront triés une seule fois au début de l'algorithme pour l'heuristique de séparation).

Remarque : si l'algorithme nous donne une solution entière : on a trouvé la solution optimale (selon les choix qui sont déjà fait).

Si l'algorithme nous donne une solution avec un terme dans $]0,1[$, la partie entière de la valeur de cette solution est une borne supérieure sur la solution optimale du problème entier.

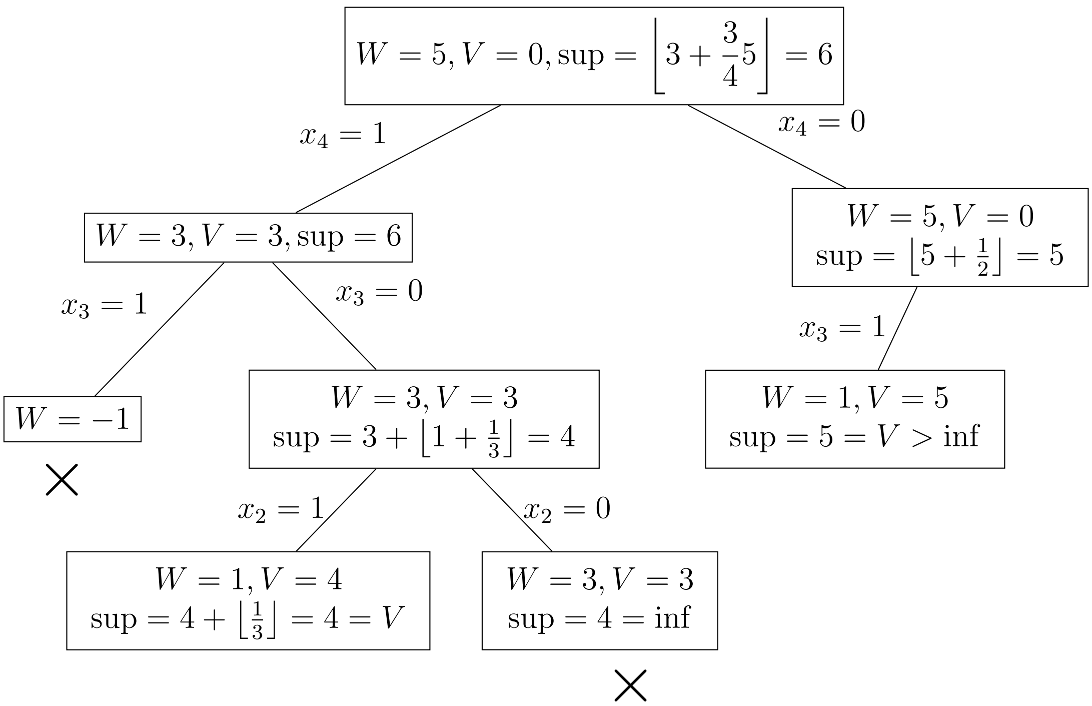
Exemple : on considère l'instance suivante :

i	1	2	3	4
p_i	1	1	5	3
w_i	3	2	4	2

$W = 5$

Le tri des objets indique qu'on doit les traiter dans l'ordre suivant : 4, 3, 2, 1.

On note au cours de l'exécution, W la capacité courante du sac à dos, et V la valeur totale courante des objets sélectionnés.



Solution $(0,1,0,1)$, et $\inf = 4$

Solution $(0,0,1,0)$, et $\inf = 5$

Conclusion : sélectionner uniquement l'objet 3 est une solution optimale.

$\boxed{\text{Exo}}$ écrire l'exécution de l'algorithme si l'heuristique de séparation consiste à prendre les objets par ordre d'indice ou par ordre de poids décroissant ou de valeur décroissante.

Tester aussi l'heuristique d'évaluation qui consiste à prendre la borne des valeurs des objets comme borne supérieure.

1.2 Optimisation et NP-complétude

1.2.1 Introduction

On considère un problème d'optimisation caractérisé par une relation $\mathcal{R} \subseteq A \times B$ et une fonction de coût $c : B \longrightarrow \mathbb{R}^+$. On rappelle que le problème de décision associé à ce problème d'optimisation s'énonce ainsi : étant donné une instance $a \in A$ et un entier $k \in \mathbb{N}$, existe-t-il une solution $b \in B$ telle que

$$\begin{cases} a\mathcal{R}b \\ c(b) \leqslant k \end{cases}$$

Remarque : on peut aussi considérer des problèmes de maximisation et la condition à satisfaire est alors $c(k) \geqslant k$.

Fait : s'il existe un algorithme de complexité polynomiale qui résout le problème d'optimisation, alors le problème de décision associé appartient à la classe P. En effet, il suffit pour une instance a d'exécuter l'algorithme qui résout le problème d'optimisation sur a (complexité polynomiale en la taille de a) puis de comparer le coût de la solution optimale obtenue et k (complexité $\mathcal{O}(\log k)$). Cela donne un algorithme polynomial qui résout le problème de décision.

Conséquence : si le problème de décision associé à un problème d'optimisation est **NP**-complet, alors on a peu d'espoir de trouver un algorithme polynomial qui résout le problème d'optimisation.

1.2.2 Retour au problème du sac à dos

On sait que le problème en variables réelles peut être résolu en temps $\mathcal{O}(n \log n)$ par un algorithme glouton et que cet algorithme ne fonctionne pas pour le problème en variables entières.

- Résolution par programmation dynamique : on considère une instance

$$(w_1, \ldots, w_n, p_1, \ldots, p_n, W)$$

du problème du sac à dos en variables entières.

$\forall i \in [\![0\ ;\ n]\!]\ ,\ \forall w \in [\![0\ ;\ W]\!]$, on note $V(i, w)$ la valeur maximale que l'on peut atteindre en sélectionnant des objets parmi ceux d'indices 1 à i dans un sac à dos de capacité w.

On cherche à obtenir $V(n, W)$.

$\forall i \in [\![0\ ;\ n-1]\!]\ ,\ \forall w \in [\![0\ ;\ W]\!]\ ,$

$$V(i+1, w) = \begin{cases} V(i, w) & \text{si} \quad \overbrace{w_{i+1} > w}^{\text{on ne peut pas sélectionner l'objet } i+1} \\ \max(\underbrace{V(i, w)}_{\text{on ne sélectionne pas l'objet } i+1},\ \underbrace{p_{i+1} + V(i, w - w_{i+1})}_{\text{on sélectionne l'objet } i+1}) & \text{sinon} \end{cases}$$

$\forall w \in [\![0\ ;\ W]\!]\ ,\ V(0, W) = 0$

On peut donc remplir la matrice V ligne par ligne (par i croissant) et retrouver une solution réalisant $V(n, W)$ en temps $\mathcal{O}(W_n)$.
Conclusion ? Aucune car c'est un algorithme de complexité exponentielle en la taille de l'instance (W est de taille $\mathcal{O}(\log w)$).

- Proposition

> On considère le problème de décision suivant :
> Sac_a_dos : étant donné n poids $w_1, \ldots, w_n \in \mathbb{N}$, n valeurs $x_1, \ldots, x_n \in \{0,1\}$ telles que
> $$\begin{cases} \displaystyle\sum_{i=1}^{n} x_i p_i \geqslant k \\ \displaystyle\sum_{i=1}^{n} x_i w_i \leqslant W \end{cases} \qquad ?$$
> Sac_a_dos est **NP**-complet.

□

– Sac_a_dos $\in$ **NP** : $x_1, \ldots, x_n$ est un certificat vérifiable en temps polynomial.

– Sac_a_dos est **NP**-difficile : on procède par réduction de 2-Partition (*cf* TD[46]) : étant donné $S = \{a_1, \ldots, a_n\} \subseteq \mathbb{N}$, existe-t-il $I \subseteq [\![1 \ ; \ n]\!]$ tel que

$$\sum_{i \in I} a_i = \sum_{i \in [\![1 \ ; \ n]\!] \setminus I} a_i \qquad ?$$

Soit $S = \{a_1, \ldots, a_n\}$ une instance de 2-Partition.
On construit l'instance suivante de Sac_a_dos :

$$\begin{cases} \forall i \in [\![1 \ ; \ n]\!], \ w_i = p_i = a_i \\ W = k = \dfrac{1}{2}\displaystyle\sum_{i=1}^{n} a_i \end{cases}$$

Cette instance est calculable en temps polynomial en la taille de S et cela constitue une réduction :

+ Si $\exists I \subseteq [\![1 \ ; \ n]\!] \mid \displaystyle\sum_{i \in I} a_i = \sum_{i \in [\![1 \ ; \ n]\!] \setminus I} a_i$, alors

$$\sum_{i=1}^{n} a_i = \sum_{i \in I} a_i + \sum_{i \in [\![1 \ ; \ n]\!] \setminus I} a_i = 2\sum_{i \in I} a_i$$

donc en notant $\forall i \in [\![1 \ ; \ n]\!], \ x_i = \mathbb{1}_I(i)$, on obtient

$$\sum_{i=1}^{n} x_i p_i = \sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} x_i a_i = \sum_{i=1}^{n} \mathbb{1}_I(i) a_i = \sum_{i \in I} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i = k = W$$

Donc les x_i sont une solution de l'instance de Sac_a_dos associée.

+ Réciproquement, si

$$\exists(x_1,\dots,x_n)\in\{0,1\}^n \ \mid\ \begin{cases}\displaystyle\sum_{i=1}^n x_i w_i \leqslant W\\ \displaystyle\sum_{i=1}^n x_i p_i \geqslant k\end{cases}$$

Alors

$$\frac{1}{2}\sum_{i=1}^n a_i = k \leqslant \sum_{i=1}^n x_i p_i = \sum_{i=1}^n x_i a_i = \sum_{i=1}^n x_i w_i \leqslant W = \frac{1}{2}\sum_{i=1}^n a_i$$

Donc

$$\begin{aligned}\sum_{i=1}^n x_i a_i &= \frac{1}{2}\sum_{i=1}^n a_i\\ &= \frac{1}{2}\left(\sum_{\substack{i=1\\x_i=1}}^n a_i + \sum_{\substack{i=1\\x_i=0}}^n a_i\right)\\ &= \frac{1}{2}\left(\sum_{\substack{i=1\\x_i=1}}^n x_i a_i + \sum_{\substack{i=1\\x_i=0}}^n a_i\right)\\ &= \frac{1}{2}\left(\sum_{i=1}^n x_i a_i + \sum_{\substack{i=1\\x_i=0}}^n a_i\right)\end{aligned}$$

Donc $\displaystyle\sum_{\substack{i=1\\x_i=1}}^n a_i = \sum_{i=1}^n x_i a_i = \sum_{\substack{i=0\\x_i=0}}^n a_i$

Donc $I=\{i\in⟦1\ ;\ n⟧ \mid x_i=1\}$ est une solution à l'instance de 2-Partition ■

1.2.3 Remarque

Comme on a peu d'espoir de trouver efficacement une solution optimale à un problème d'optimisation dont le problème de décision associé est **NP**-complet, on va plutôt chercher efficacement une solution « pas trop mauvaise ». On cherche donc à concevoir un algorithme de complexité polynomiale qui fournit des solutions pour lesquelles on peut estimer la « distance » à l'optimum.

• Exemple : l'algorithme glouton vu en 1.1.2 (page 2) pour le problème du sac à dos est très mauvais vis à vis du problème en variables entières : si $k\in\mathbb{N}^*$ et $W\in\mathbb{N}\setminus\{0,1\}$, on considère deux objets tels que

$$\begin{cases}p_1 = 1\\ w_1 = \dfrac{W-1}{k}\end{cases} \qquad \text{et} \qquad \begin{cases}p_2 = k\\ w_2 = W\end{cases}$$

L'algorithme glouton sélectionne l'objet 1 car $\dfrac{1}{\dfrac{W-1}{k}} = \dfrac{k}{W-1} > \dfrac{k}{W}$, ce qui donne une solution de valeur 1 alors que la solution optimale consiste à prendre l'objet 2 pour une valeur k.
Ainsi $\forall k \in \mathbb{N}^*$, il existe une instance telle que la solution optimale est k fois meilleure que celle calculée par l'algorithme glouton.

• On peut faire mieux en modifiant légèrement l'algorithme : on garde la meilleure solution entre celle de l'algorithme glouton et celle qui consiste à ne prendre que l'objet de valeur maximale.

Proposition :

> On note pour toute instance e, $V^*(e)$ la valeur d'une solution optimale, et $V(e)$ la valeur de la solution calculée par l'algorithme glouton modifié.
> Alors
> $$\forall e,\ V^*(e) \leqslant 2V(e)$$

□

On note $e = (w_1, \ldots, w_n, p_1, \ldots, p_n, W)$.

Quitte à renuméroter, on peut supposer que les objets sont tirés par $\dfrac{p_i}{w_i}$ décroissant.

On sait que toute solution entière donne une valeur inférieure à celle d'une solution optimale réelle. De plus, une telle solution est calculée *via* l'algorithme glouton.

Alors, en notant $j \in [\![1\ ;\ n]\!]$ l'indice du premier objet que l'on ne peut pas placer intégralement dans le sac à dos, on a

$$\begin{aligned}
V^*(e) &\leqslant \sum_{i=1}^{j-1} p_i + \underbrace{\frac{W - \sum_{i=1}^{j-1} w_i}{w_j}}_{<1} p_j \\
&\leqslant \sum_{i=1}^{j-1} p_i + p_i \\
&\leqslant \sum_{i=1}^{j-1} p_i + \max_{i \in [\![1\ ;\ n]\!]} p_i \\
&\leqslant 2 \max\left(\sum_{i=1}^{j-1} a_i,\ \max_{i \in [\![1\ ;\ n]\!]} p_i\right)
\end{aligned}$$

(L'algorithme glouton prend tous les objets de 1 à $j-1$ puis la fraction de l'objet j qui permet de remplir le sac à dos) ■

Remarque : on a donc un algorithme de même complexité que l'algorithme glouton et tel que la solution calculée est toujours de valeur supérieure à la moitié de la valeur optimale.

$\boxed{\text{H.P}}$ $\forall \varepsilon \in \mathbb{R}_+^*$, il existe un algorithme de complexité $\mathcal{O}\left(\dfrac{1+\varepsilon}{\varepsilon} n^3\right)$ déterminant une solution entière au problème du sac à dos et tel que $\forall e$ instance, la valeur $V(e)$ de la solution calculée vérifie $V^*(e) \leqslant (1+\varepsilon)V(e)$.

Idée : par programmation dynamique : $\forall i \in [\![0\ ;\ n]\!]$, $\forall p \in \left[\!\!\left[0\ ;\ \sum_{i=1}^{n} p_i\right]\!\!\right]$, on calcule le poids minimal $W(i,p)$ réalisable en sélectionnant des objets parmi ceux d'indices de 1 à i de sorte que la valeur obtenue vaille p (et le poids $\leqslant W$).

On le fait avec des objets de valeur modifiée en fonction de n, ε, $p_{\max} = \max_{i \in [\![1\ ;\ n]\!]} p_i$

$\forall i \in [\![1\ ;\ n]\!]$, on note $\left\lfloor \dfrac{p_i}{2^t} \right\rfloor$, où

$$t = \left\lfloor \log\left(\frac{\varepsilon}{1+\varepsilon}\frac{p_{\max}}{n}\right)\right\rfloor$$