Chapitre 17 : Complément d'algorithmique

Table des matières

List of Algorithms

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

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

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).

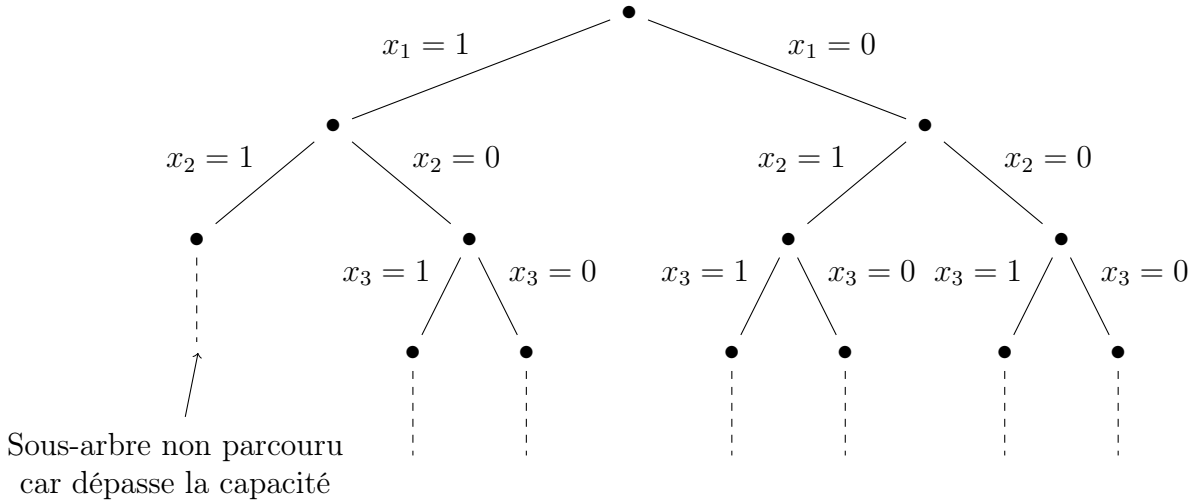
On peut parfois accélérer la recherche grâce à l'heuristique du retour sur trace (*cf* chapitre 8, 4.3).

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 $\frac{p_i}{w_i}$ 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.

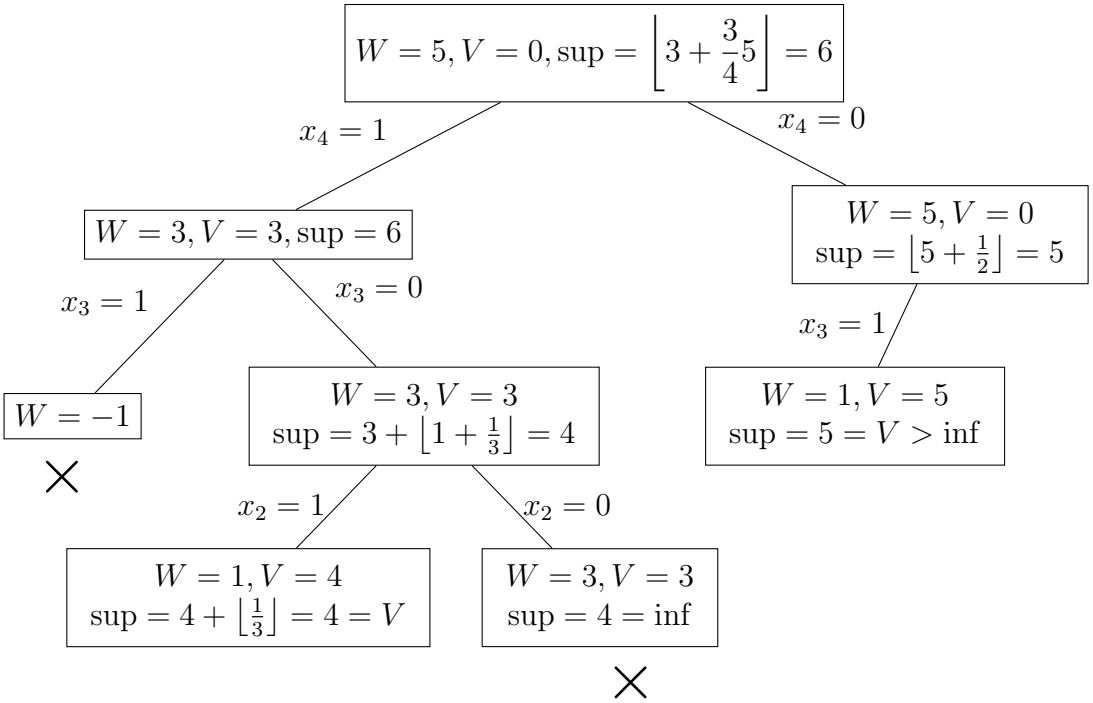
Exemple : on considère l'instance suivante :

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.

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

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

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

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 $w_1, \dots, w_n \in \mathbb{N}$, n valeurs $x_1, \dots, x_n \in \{0, 1\}$ telles que
> $$\begin{cases} \sum_{i=1}^{n} x_i p_i \geqslant k \\ \sum_{i=1}^{n} x_i w_i \leqslant W \end{cases} \quad ?$$
> SAC_A_DOS est **NP**-complet.

□

– SAC_A_DOS $\in$ **NP** : $x_1, \dots, x_n$ 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 $S = \{a_1, \dots, a_n\} \subseteq \mathbb{N}$, existe-t-il $I \subseteq [\![1\ ;\ n]\!]$ tel que

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

Soit $S = \{a_1, \dots, a_n\}$ une instance de 2-PARTITION.
On construit l'instance suivante de SAC_A_DOS :

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

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

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

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

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 $e = (w_1, \dots, w_n, p_1, \dots, p_n, W)$.

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.

H.P $\forall \varepsilon \in \mathbb{R}_+^*$, il existe un algorithme de complexité $\mathcal{O}\left(\frac{1+\varepsilon}{\varepsilon} n^3\right)$ 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 $\left\lfloor \frac{p_i}{2^t} \right\rfloor$, où

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

1.3 Algorithmes d'approximation

1.3.1 Définition (*algorithme d'approximation*)

On considère une problème d'optimisation caractérisé par une relation $\mathcal{R} \subseteq A \times B$ et une fonction de coût $c : B \longrightarrow \mathbb{R}^+$.

Soit $\alpha \in \mathbb{R}_+^*$, et M un algorithme tel que $\forall a \in A$, M appliqué à a termine et revoie une solution associée à l'instance a, *i.e* $a\mathcal{R}M(a)$.

On dit que M est un *algorithme d'approximation* de facteur α, ou *α-approximation*, pour le problème d'optimisation si, en notant $\forall a \in A$, $c^*(a)$ le coût d'une solution optimale associée à a :

- si on a un problème de minimisation, $\forall a \in A$, $c(M(a)) < \alpha c^*(a)$ (et $\alpha > 1$)
- si on a un problème de maximisation, $\forall a \in A$, $c(M(a)) \geqslant \alpha c^*(a)$ (et $\alpha < 1$).

1.3.2 Exemple

L'algorithme glouton pour le problème du sac à dos n'est pas un algorithme d'approximation car on a vu en 1.2.3 (page 8) que l'on peut construire des instances telles que le rapport entre la valeur de la solution calculée est valeur optimale est arbitrairement petit.

En revanche, on a vu que l'algorithme modifié constitue une $\frac{1}{2}$–approximation.

1.3.3 Remarques

• On parle parfois d'$\alpha(n)$–approximation, où le facteur α est variable et dépend de la taille de l'instance.

• On utilise souvent des algorithmes gloutons pour construire des algorithmes d'approximation.

1.3.4 Exemple

On considère le problème d'optimisation suivant, associé à VERTEX_COVER : étant donné un graphe $G = (S, A)$, déterminer une couverture par les sommets de G de taille minimale.

On a peu d'espoir de résoudre en temps polynomial ce problème d'optimisation car VERTEX_COVER est **NP**–complet (*cf* chapitre 16, 2.3.5) : si c'était possible, étant donné une instance (G, k) de VERTEX_COVER, on pourrait déterminer une couverture optimale de G et comparer sa taille à k (si sa taille est inférieure à k, on peut toujours ajouter des sommets pour obtenir une couverture de taille k).

• Algorithme d'approximation :

Idée : on pourrait sélectionner les extrémités d'un ensemble d'arêtes qui rencontrent toutes les autres arêtes du graphe. Pour minimiser le nombre de sommets sélectionnés, on cherchera des arêtes sans extrémité commune, donc sur un couplage maximal. On ne cherchera pas un couplage maximum car on veut minimiser le nombre de sommets, donc d'arêtes.

D'où l'algorithme glouton :

Algorithm 2: Approximation pour VERTEX_COVER

```
1 C ← ∅;
2 while A ≠ ∅ do
3 |   Extraire une arête {s, t} de A;
4 |   C ← {s, t} ∪ C;
5 |   Supprimer les arêtes adjacentes à s ou à t;
6 return C;
```

• Proposition :

> Cet algorithme est une 2–approximation pour le problème de la couverture par les sommets minimale.

□

On démontre d'abord que l'ensemble C calculé est une couverture par les sommets de G.

Pour cela, on démontre l'invariant : « toutes les arêtes supprimées sont couvertes par les sommets de C ».

En fin d'exécution, toutes les arêtes ont été supprimées, donc sont couvertes.

On note M l'ensemble des arêtes extraites sur la ligne 3 de l'algorithme 2.

Par construction, $|C| = 2\,|M|$.

De plus, M est un couplage de G : on démontre l'invariant : « M est un couplage, et toutes les arêtes de A sont non adjacentes à celles de M ».

– Initialement, $M = \varnothing$, donc l'invariant est vrai.

– Invariance : on suppose l'invariant vrai et on note $\{s, t\}$ la prochaine arête

extraite de A.

Par l'invariant, $\{s,t\}$ est non adjacente aux arêtes de M et M est un couplage.

Après suppression des arêtes incidentes à s ou à t, il ne reste dans A que des arêtes non adjacentes à $\{s,t\}$ et non adjacentes aux arêtes de M d'après l'invariant.

L'invariant reste donc vrai.

On note maintenant C^* une couverture optimale.

Comme les sommets de C^* doivent couvrir les arêtes de M, et comme M est un couplage, il y a nécessairement au moins un sommet dans C^* pour chaque arête de M.

Donc

$$|C^*| \geqslant |M| = \frac{|C|}{2}$$

i.e

$$|C| \leqslant 2\,|C^*|$$

■

• Remarque : si on suppose que G est un arbre, alors il existe un algorithme glouton optimal pour ce problème.

Comme il faut couvrir l'unique arête incidente à une feuille, on peut sélectionner l'autre extrémité de cette arête en espérant couvrir ainsi plusieurs arêtes. Supprimer les arêtes incidentes à ce sommet peut créer plusieurs arbres, donc il faudra tenir compte du fait que le graphe peut devenir une forêt.

D'où l'algorithme suivant :

Algorithm 3: Algorithme optimal pour VERTEX_COVER sur les arbres

```
1  F ← file de priorité min dont les éléments sont les sommets, dont la priorité
     est leur degré;
2  C ← ∅;
3  while F ≠ ∅ do
4      Extraire l'élément s de priorité min de F;
5      if s admet un voisin t then
6          C ← C ∪ {A};
7          A ← A \ {{s, t}};
8          for chaque voisin u de t do
9              A ← A \ {{t, u}};
10             Décrémenter la priorité de u dans F;
11         Mettre la priorité de t dans F à 0;
12     return C;
```

Complexité :

$$\mathcal{O}\left(\underbrace{|S|+|A|}_{\text{création de } F} + \sum_{s\in S} \underbrace{\log|S|}_{\substack{\text{extraction}\\ \text{du min}}} + \sum_{t\in S} \underbrace{(d(t)+1)\log|S|}_{\substack{\text{mises à jour}\\ \text{de priorité}}}\right) = \mathcal{O}((|S|+|A|)\log|S|)$$

Ce qui est polynomial en $|G|$.

• Proposition :

> Chaque sommet extrait est de degré 0 ou 1 au moment de l'extraction.

□

On a l'invariant « les priorité des sommets sont leurs degrés ».
Il suffit alors de démontrer l'invariant « le sous-graphe G_F induit par F est une forêt » car une sommet de degré minimal dans une forêt est soi isolé, soit une feuille d'un arbre.
– Initialement, $G_F = G$ qui est un arbre, donc une forêt.
– Invariance : si G_F est une forêt, et si on extrait s de F, alors $d(s) \in \{0, 1\}$.
Si $d(s) = 0$, on a juste retiré un sommet isolé, donc G_F reste une forêt.
Si $d(s) = 1$ et t est l'unique voisin de s, on décompose l'arbre de G_F contenant s et t en un sommet isolé (A) et autant de sous-arbres que t admet de voisins différents de s.
Donc G_F reste une forêt. ■

• Proposition

> L'algorithme glouton calcule une couverture par les sommets optimale de G.

□

Tout d'abord, C est bien une couverture car chaque arête est supprimée parce qu'elle est couverte par un élément ajouté à C. Toutes les arêtes sont bien supprimées car on retire teous les sommets de la file et pour chaque sommet extrait, il est de degré 0 ou 1 lors de son extraction, *i.e* toutes els arêtes incidentes à ce sommet ont été supprimées sauf peut-être une qui sera supprimé au moment de l'exécution.

On démontre l'optimalité grâce à l'invariant « il existe une couverture optimale C^* telle que $C \subseteq C^*$ ».
– Initialement, $C = \varnothing$, et il existe une couverture optimale donc l'invariant est vrai.
– Invariance : on suppose qu'il existe une couverture optimale $C^* \mid C \subseteq C^*$, et on note s le sommet extrait.
Si $d(s) = 0$, C ne change pas et C^* convient encore.
Sinon, $d(s) = 1$, et on note t l'unique voisin de s.
+ si $t \in C^*$, C^* convient encore.
+ sinon, $s \in C^*$ car $\{s, t\}$ doit être couverte.
On montre alors que $(C^* \setminus \{s\}) \cup \{t\}$ est une couverture optimale contenant $C \cup \{t\}$:
$s \notin C$, car sinon, au moment de son extraction, $d(s) = 0$, et on a supposé $d(s) = 1$.
Donc $C \subseteq C^* \setminus \{s\}$, et $C \cup \{t\} \subseteq (C^* \setminus \{s\}) \cup \{t\}$.
Le cardinal ne change pas, donc il reste à démontrer que $(C^* \setminus \{s\}) \cup \{t\}$ est une

couverture.

C'est vrai car la seule arête incidente à s qui n'est pas couverte par C, donc par $C^* \setminus \{s\}$ est l'arête $\{s,t\}$, qui est couverte par $\{t\}$. Les autres arêtes sont bien couvertes par $C^* \setminus \{s\}$.

Finalement, en sortie d'algorithme, C est une couverture incluse dnas une couverture optimale, donc c'est une couverture optimale. ■

1.3.5 Exemple

On transforme le problème de décision SAT en le problème d'optimisation MAX_SAT : étant donné une formule φ sous FNC, déterminer une valuation qui maximise le nombre de clauses de φ satisfaites.

• Remarque : le problème de décision associé à MAX_SAT est **NP**–complet (réduction de SAT avec le nombre de clauses de la formule comme seuil).

On peut de même considérer les problèmes MAX-k-SAT, pour $k \geqslant 1$, où les formules considérées sont telles que toutes leurs clauses sont de taille $\leqslant k$.

• Proposition

> MAX-k-SAT est **NP**-complet $\forall k \geqslant 2$.

□

Il suffit le de démontrer pour $k = 2$ car la fonction identité est une réduction polynomiale de MAX-k-SAT $\forall k \geqslant 2$, en remarquant que tous ces problèmes appartiennent à la classe **NP** (certificat = valuation, on compte le nombre de clauses satisfaites).

On montre que 3-SAT $\leqslant_{\mathrm{p}}$ MAX-2-SAT

Soit φ une formule sous forme normale conjonctive dont les clauses sont de taille au plus trois. On peut supposer que toutes les clauses de φ sont de taille exactement trois, quitte à répéter une littéral dans les clauses trop petites.

On écrit donc

$$\varphi = \bigwedge_{i=1}^{k} (a_i \vee b_i \vee c_i)$$

avec a_i, b_i, c_i des littéraux.

On constuit alors l'instance suivante de MAX-2-SAT :

$$\begin{aligned} \text{Formule} &: \bigwedge_{i=1}^{k} \psi_i \\ \text{Seuil} &: 7k \end{aligned}$$

où :

$$\psi_i = (a_i \wedge b_i \wedge c_i \wedge d_i \wedge (\neg a_i \vee \neg b_i) \wedge (\neg a_i \vee \neg c_i) \wedge (\neg b_i \vee \neg c_i) \wedge (a_i \vee \neg d_i) \wedge (b_i \vee \neg d_i) \wedge (c_i \vee \neg d_i))$$

Si v est une modèle de φ, alors $\forall i \in [\![1\ ;\ k]\!]$, il y a entre 1 et 3 littéraux de la clause $a_i \vee b_i \vee c_i$ qui sont satisfaits par v. Dans tous les cas, on peut choisir la valeur de d_i de sorte que 7 des 10 clauses construites à partir de cette clause soient satisfaites.

Au total, on aura bien au moins $7k$ clauses satisfaites.

Réciproquement, si une valuation v satisfait au moins $7k$ clauses de la formule construite : $\forall i \in [\![1\ ;\ k]\!]$, si aucun des littéraux a_i, b_i, c_i n'est satisfait par v, alors on ne peut satisfaire qu'au plus six clauses parmi des dix associés à ces littéraux.

Si un, deux, ou trois littéraux parmi a_i, b_i, c_i sont satisfait par v, alors comme avant par disjonction de cas, on montre qu'au plus sept clauses parmi les dix associés à ces littéraux peuvent être satisfaites. Donc pour chaque clause de φ il y a au moins un littéral satisfait par v. ■

• Algorithme d'approximation pour MAX_SAT : on propose d'utiliser l'algorithme naïf suivant :

Algorithm 4: Approximation pour MAX_SAT

```
for chaque variable x de φ do
    Choisir la valeur de vérité de x par un tirage aléatoire uniforme dans {V, F};
return la valuation obtenue;
```

On parle d'*algorithme probabiliste* ou *randomisé* car on effectue des tirages aléatoires. Le résultat de cet algorithme dépend des tirages effectués, donc pour évaluer la qualité de cet algorithme en terme d'approximation, on compare la valeur d'une solution optimale avec l'espérance de la valeur du résultat de cet algorithme.

On note S la variable aléatoire du nombre de clauses satisfaites par le résultat de l'algorithme, et $\forall c$ clause de φ, on note S_c la variable aléatoire qui vaut 1 si c est satisfaite par le résultat de l'algorithme, et 0 sinon.

On sait que $S = \sum_{c \in \varphi} S_c$, donc par linéarité,

$$E(S) = \sum_{c \in \varphi} E(S_c) = \sum_{c \in \varphi} \mathbb{P}(S_c = 1)$$

Soit c une clause de φ :

– Si c contient une variable et sa négation, $\mathbb{P}(S_c = 1) = 1$

– Sinon, on note k_c le nombre de littéraux indépendants de c, *i.e* portant sur des variables différentes. c n'est satisfaite si et seulement si tous les littéraux prennent la valeur F, ce qui arrive avec la probabilité $\frac{1}{2^{k_0}}$ car les tirages aléatoires sont indépendants.

Donc

$$\mathbb{P}(S_c = 1) = 1 - \frac{1}{2^{k_0}} \geqslant \frac{1}{2}$$

Donc $E(S) \geqslant \frac{k}{2} \geqslant \frac{V^*(\varphi)}{2}$, où $V^*(\varphi)$ est le nombre maximal de clauses que l'on peut satisfaire dans φ (majoré par le nombre de clauses de φ).

Cet algorithme est donc une $\frac{1}{2}$–approximation.

Remarque : si toutes les clauses sont de taille $\geqslant 2$, on obtient une $\frac{3}{4}$–approximation.

2 Algorithmes probabilistes

2.1 Généralités

2.1.1 Définition (*algorithme probabiliste*)

Un *algorithme probabiliste* est un un algorithme au sens de chap.16, 2.1.2, qui donne les opérations élémentaires possibles, en ajoutant une opération de tirage aléatoire correspondant au tirage aléatoire uniforme d'un bit.

On dit que l'algorithme est *déterministe* s'il n'utilise pas l'opération de tirage aléatoire.

2.1.2 Remarque

Deux exécutions d'un algorithme probabiliste peuvent donner des résultats différents sur le même paramètre (exemple : algorithme 4, page 15) ou bien effectuer un nombre différent d'opérations élémentaires (exemple : 2.1.3, page 16).

La complexité temporelle pour une entrée fixée est donc une variable aléatoire dont on étudie en général l'espérance.

2.1.3 Exemple : le tri rapide randomisé

Idée : pour éviter de tomber dans le pire cas du tri rapide, on tire aléatoirement uniformément le pivot parmi les éléments à trier.

Remarque : dans le pire cas, on tire par exemple systématiquement le plus petit élément, ce qui donne $\mathcal{O}(n^2)$ comparaisons, mais cela arrive avec une probabilité de $\frac{1}{n!}$.

On utilise l'algorithme suivant :

Algorithm 5: Tri rapide randomisé

```
1 Function Tri_rapide(l):
2     if |l| ⩽ 1 then
3         return l;
4     else
5         p ← élément aléatoire uniforme de l;
6         l_<, l_> ← partition de l suivant p;
7         return Tri_rapide(l_<) @ [p] @ Tri_rapide(l_>);
```

On note $l = [l_1, \ldots, l_n]$, et C la variable aléatoire du nombre de comparaisons effectuées par Tri_rapide.

On remarque que lorsque deux éléments sont comparés, c'est que l'un des deux est le pivot courant, donc ils ne seront plus comparés par la suite.

Ainsi,

$$C = \sum_{i=1}^{n} \sum_{j=i+1}^{n} C_{i,j}$$

où $C_{i,j}$ est la variable aléatoire indicatrice qui vaut 1 si l_i et l_j sont comparés, et 0 sinon.

Donc

$$E(C) = \sum i = 1^n \sum_{j=i+1}^{n} E(C_{i,j}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}$$

en notant $p_{i,j}$ la probabilité de l'événement « l_i et l_j sont comparés ».

On souhaite donc déterminer les $p_{i,j}$.

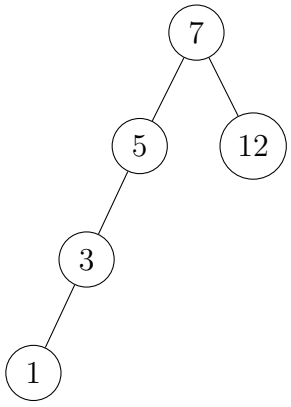
On peut représenter une exécution de cet algorithme par un arbre binaire : la racine est le pivot tiré et les sous-arbres gauche et droit sont les arbres représentant les exécutions des appels récursifs sur $l_<$ et $l_>$.

Exemple : $[1, 12, 3, 7, 5]$

Une exécution possible :

- on tire 7, $l_< = [1, 3, 5]$, $l_> = [12]$
- On trie $l_<$:
  - on tire 5. $l_< = [1, 3]$, $l_> = []$.
  - On trie $l_<$:
    - on tire 3. $l_< = [1]$, $l_> = []$

L'arbre associé :



Remarque : la racine est comparée avec tous les éléments des deux sous-arbres et les éléments du sous-arbre gauche ne sont jamais comparés avec ceux du sous-arbre droit.

Ainsi, si $1 \leqslant i < j \leqslant n$, il y a deux cas :

- $\exists k \in [\![1\ ;\ n]\!] \mid l_i < l_k < l_j$ et l_k est choisi come pivot courant juste avant l_i et l_j : l_i et l_j se retrouvent alors dans des sous-arbres distincts et ne sont pas comparés.

– $\forall k \in [\![1\ ;\ n]\!] \mid l_i < l_k < l_j$, l_k est choisi comme pivot après l_i ou l_j, donc soit l_i, soit l_j est choisi comme pivot en premier parmi tous ces éléments donc soit l_i, soit l_j est racine du sous-arbre associé et l_i et l_j sont comparés.

En renumérotant les l_i par ordre croissant, nommés l'_i, on sait que l'_i et l'_j sont comparés (avec $i < j$) si et seulement si l'_i ou l'_j est le premier élément choisi comme pivot parmi les $l'_k,\ k \in [\![i\ ;\ j]\!]$.

comme les tirages aléatoires sont uniformes, la probabilité que l'_k soit tiré comme pivot en premier est la même quelle que soit la valeur de k.

Donc la probabilité $p'_{i,j}$ que l'_i et l'_j soient comparés vaut $\dfrac{2}{j-i+1}$

Donc

$$\begin{aligned} E(C) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} p_{i,j} = \sum_{i=1}^{n}\sum_{j=i+1}^{n} p'_{i,j} = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &\leqslant 2\sum_{i=1}^{n}\underbrace{\sum_{k=1}^{n}\frac{1}{k}}_{H_n} \\ &\leqslant 2nH_n \end{aligned}$$

Or $H_n = \ln n + \gamma + o(1)$.
Donc $E(C) = \mathcal{O}(n \log n)$.

Remarque : on peut démontrer qu'avec une forte probabilité la valeur de C n'est pas très éloignée de son espérance, donc cet algorithme est très souvent optimal en terme d'ordre de grandeur du nombre de comparaisons.