Algorithmes d'approximation

Que faire lorsqu'on doit résoudre un problème $\mathcal{NP}$–dur? On peut développer des algorithmes qui ne donnent pas forcément l'optimum. Il peut alors être intéressant de borner l'erreur commise par l'algorithme. En général, si Opt(I) et A(I) correspondent à la valeur optimale de l'instance I et à la valeur produite par l'algorithme A, on s'intéressera à borner la différence Opt(I)-A(I) ou le ratio A(I)/Opt(I).

Quelques observations

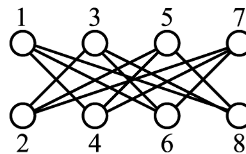
Considérons le problème consistant à déterminer le plus petit nombre de couleurs nécessaires pour colorer les sommets d'un graphe, de telle sorte qu'aucune arête n'ait ses deux extrémités de même couleur. On a vu que ce problème est $\mathcal{NP}$–dur.

Une coloration des sommets d'un graphe G=(V,E) peut être vue comme une fonction $c : V \rightarrow \mathbb{N}$ telle que $c(i) \neq c(j)$ si $\{i,j\} \in E$. L'algorithme ci-dessous détermine une coloration qui n'est pas forcément minimale.

Coloration Séquentielle

Pour i=1 à n **faire**
    Donner au sommet i la plus petite couleur n'apparaissant pas dans son voisinage.

Appliquons cet algorithme au graphe G=(V,E) où V={1,…,2n} et chaque sommet i pair est relié à tous les sommets j impairs , sauf i+1. Par exemple, pour n=4, on obtient le graphe suivant :



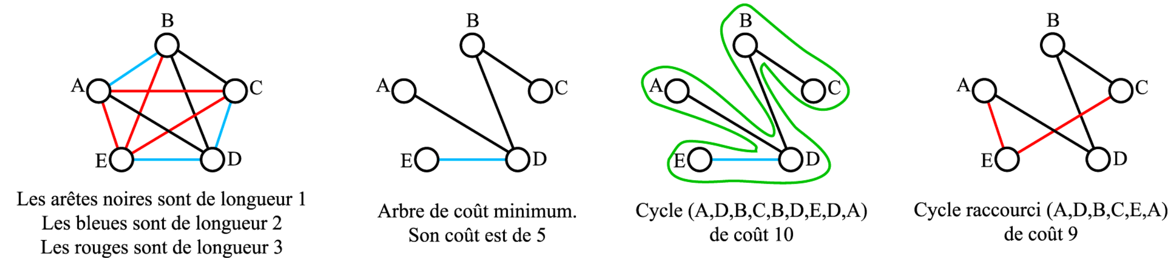
L'algorithme utilisera n couleurs alors que 2 suffisent. En notant Opt(G) le nombre minimum de couleurs nécessaires et A(G) le nombre de couleurs utilisées par l'algorithme ci-dessus, on a donc :

$$\frac{A(G)}{Opt(G)} = \frac{n}{2}$$

Considérons le problème du voyageur de commerce qui consiste à déterminer un cycle hamiltonien de longueur minimale dans un graphe complet G. Supposons que la matrice des distances satisfait l'inégalité triangulaire, c'est-à-dire que $d_{xz} \leq d_{xy} + d_{yz}$ pour tout triplet x,y,z de sommets. Ce problème est $\mathcal{NP}$–dur. On peut cependant déterminer une solution non optimale à l'aide de l'algorithme suivant qui est illustré ci-dessous.

Minimum spanning tree

Déterminer un arbre de coût minimal dans G;
Déterminer un cycle qui passe exactement 2 fois par chaque arête de l'arbre (et nulle part ailleurs);
Emprunter des raccourcis pour éviter de passer plusieurs fois par un même sommet;



Les arêtes noires sont de longueur 1
Les bleues sont de longueur 2
Les rouges sont de longueur 3

Arbre de coût minimum.
Son coût est de 5

Cycle (A,D,B,C,B,D,E,D,A)
de coût 10

Cycle raccourci (A,D,B,C,E,A)
de coût 9

Notons Opt(G) la longueur du plus petit cycle hamiltonien dans G et A(G) la longueur du cycle produit par cet algorithme. Comme l'arbre de coût minimum est de coût inférieur ou égal à Opt(G), On a

$$\frac{A(G)}{Opt(G)} \leq 2$$

Considérons le **problème du sac à dos** qui peut être décrit comme suit. On dispose de n objets $O_1,\ldots,O_n$. Chaque objet O_i a un poids w_i et une valeur v_i. On veut maximiser la valeur du chargement du sac sous la contrainte que le poids des objets dans le sac ne dépasse pas une valeur donnée W. On doit donc résoudre :

$$\begin{cases} \text{Max } \sum_{i=1}^{n} x_i v_i \\ \text{s.c. } \sum_{i=1}^{n} x_i w_i \le W \\ \quad 0 \le x_i \le 1 \text{ entiers} \end{cases}$$

Si on relaxe la contrainte d'intégralité, la solution optimale est facile à déterminer. Il suffit renuméroter les objets de telle sorte que $v_i/w_i \ge v_j/w_j$ pour tout $i<j$, et de poser ensuite

$$x_1=\ldots=x_r=1, \qquad x_{r+1}=\frac{W-\sum_{i=1}^{r} w_i}{w_{r+1}} \qquad \text{et} \qquad x_{r+2}=\ldots=x_n=0.$$

où r est le plus grand indice tel que $\sum_{i=1}^{r} w_i \le W$. En rajoutant les contraintes d'intégralité, le problème devient $\mathcal{NP}$–dur. On peut cependant déterminer une solution non optimale à l'aide de l'algorithme suivant

Algorithme Sac_À_Dos_1

Renuméroter les objets de telle sorte que $v_i/w_i \ge v_j/w_j$ pour tout $i<j$. Poser Poids :=0;
Pour i=1 à n **faire**
Si Poids+$w_i \le W$ alors mettre O_i dans le sac et poser Poids := Poids + w_i

Appliquons cet algorithme au cas où n=2, W=x, $w_1=1$, $w_2=x$, $v_1=2$ et $v_2=x$. L'algorithme mettra O_1 dans le sac pour un chargement de valeur 2, alors qu'on aurait pu mettre O_2 pour un chargement de valeur x. En d'autres termes, si Opt(X) est la valeur optimale d'un chargement pour une instance X et si A(X) est la valeur du chargement produit par l'algorithme *Sac_À_Dos_1*, on a, pour cette instance X :

$$\frac{A(X)}{Opt(X)}=\frac{2}{x}$$

Soit O_{max} l'objet de plus grande valeur, et soit v_{max} sa valeur. Considérons l'algorithme suivant

Algorithme Sac_À_Dos_2

Appliquer l'algorithme *Sac_À_Dos_1*;
Si $A(X)<v_{max}$ alors mettre O_{max} dans le sac, sinon choisir la solution produite par l'algorithme *Sac_À_Dos_1*;

Soit B(X) la valeur du chargement produit par cet algorithme pour une instance X. Sur l'exemple précédent, on a B(X)=x=Opt(X). Étudions le comportement de cet algorithme en général.

En supposant $v_i/w_i \ge v_j/w_j$ pour tout $i<j$, soit r le plus petit indice tel que $W < \sum_{i=1}^{r} w_i = W'$.

Soit Opt'(X) la solution du problème du sac à dos dans lequel on remplace W par W'. Opt'(X) est donc la valeur optimale du problème suivant

$$\begin{cases} \text{Max } \sum_{i=1}^{n} x_i v_i \\ \text{s.c. } \sum_{i=1}^{n} x_i w_i \le W' \\ \quad 0 \le x_i \le 1 \text{ entiers} \end{cases}$$

On a bien sûr Opt'(X) ≥ Opt(X). En relaxant les contraintes d'intégralité, on a vu que l'optimum consiste à poser $x_1=\ldots=x_r=1$ et $x_{r+1}=\ldots=x_n=0$. Cette solution est entière et est donc également la solution optimale du problème non relaxé. On a donc Opt'(X)= $\sum_{i=1}^{r} v_i$. Notons que $A(X) \ge \sum_{i=1}^{r-1} v_i$. On a finalement :

$$B(X)=\max\{A(X),v_{max}\} \ge \frac{A(X)+v_{max}}{2} \ge \frac{\sum_{i=1}^{r-1} v_i + v_{max}}{2} \ge \frac{\sum_{i=1}^{r-1} v_i + v_r}{2} = \frac{\sum_{i=1}^{r} v_i}{2} = \frac{Opt'(X)}{2} \ge \frac{Opt(X)}{2}$$, et donc

$$\frac{B(X)}{Opt(X)} \ge \frac{1}{2}$$

Étant donnés des objets $O_1, \ldots, O_n$ de poids $w_1,\ldots,w_n$ et des sacs de capacité W, considérons les deux problèmes suivants de **remplissage de sacs (bin packing)** .

$$\begin{cases} \text{Max} & \sum_{i=1}^{n}\sum_{j=1}^{k} x_{ij} \\ \text{s.c.} & \sum_{i=1}^{n} x_{ij} w_i \leq W \quad j=1,\ldots,k \\ & \sum_{j=1}^{k} x_{ij} \leq 1 \quad i=1,\ldots,n \\ & x_{ij} \geq 0 \ \text{ entiers} \end{cases} \qquad \begin{cases} \text{Min} & \sum_{j=1}^{n} y_j \\ \text{s.c.} & \sum_{i=1}^{n} x_{ij} w_i \leq W y_j \quad j=1,\ldots,n \\ & \sum_{j=1}^{n} x_{ij} = 1 \quad i=1,\ldots,n \\ & y_j = 0 \ \text{ ou } \ 1 \\ & x_{ij} \geq 0 \ \text{ entiers} \end{cases}$$

Le 1er problème consiste à déterminer le nombre maximum d'objets qu'on peut ranger dans un nombre fixé k de sacs. Le 2ème problème s'intéresse au nombre minimum de sacs nécessaires pour ranger tous les objets. Dans tout ce qui suit, nous supposerons que les objets sont classés de telle sorte que $w_i \leq w_j$ si $i<j$.

Pour k=1, la solution optimale du 1er problème est de prendre $O_1,\ldots,O_r$ avec $\sum_{i=1}^{r} w_i \leq W$ et $\sum_{i=1}^{r+1} w_i > W$. Pour k=2, on peut utiliser l'algorithme suivant, qui n'est pas nécessairement optimal.

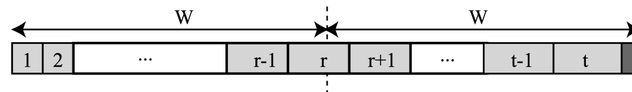
Algorithme bin-packing_1

Poser $Poids_1$:=0 et $Poids_2$:=0;
Pour i=1 à n **faire**
  Si $Poids_1+w_i \leq W$ **alors** mettre O_i dans le 1er sac et poser $Poids_1 := Poids_1+w_i$;
  Sinon si $Poids_2+w_i \leq W$ alors mettre O_i dans le 2ème sac et poser $Poids_2 := Poids_2 + w_i$;

Soit Opt(X) la solution optimale d'une instance X de ce problème. Considérons le problème de sac à dos avec les mêmes objets, avec un sac de capacité 2W, et en supposant que tous les objets sont de valeur $v_i=1$. Soit t la valeur d'une solution optimale de ce problème de sac à dos. On a $t \geq Opt(X)$ car toute solution admissible pour le problème de bin packing l'est également pour ce problème de sac à dos. De plus,

$$\sum_{i=1}^{t} w_i \leq 2W \quad \text{et} \quad \sum_{i=1}^{t+1} w_i > 2W$$

Soit r le plus petit indice tel que $\sum_{i=1}^{r} w_i > W$. On a $\sum_{i=r+1}^{t} w_i < W$. (voir figure ci-dessous).



L'algorithme ci-dessus mettra donc les r-1 premiers objets dans le premier sac. Pour ce qui est du 2ème sac, on pourra au moins y mettre les objets $O_r,\ldots,O_{t-1}$ car $\sum_{i=r}^{t-1} w_i \leq \sum_{i=r+1}^{t} w_i < W$.

En notant A(X) la valeur de la solution produite par l'algorithme ci-dessus, on a $A(X) \geq r-1+(t-r)= t-1$, et donc

$$Opt(X) - A(X) \leq 1$$

Pour le 2ème problème (minimisation du nombre de sacs), on peut étendre l'algorithme ci-dessus comme suit.

Algorithme bin-packing_2

Poser Poids :=0; nbsacs :=1;
Pour i=1 à n **faire**
  Si $Poids_1 + w_i \leq W$ **alors** poser Poids := Poids + w_i ;
  Sinon poser Poids := w_i et nbsacs :=nbsacs + 1;

Considérons le cas où n=2m pour un entier m pair et W=2c pour un entier c>3(m-1). Supposons que les objets ont des poids c-m, c-m+1, …., c-1, c+1, …, c+m. Soit Opt(X) le nombre minimum de sacs nécessaire pour ranger ces 2m objets. On a Opt(x)=m car on peut mettre l'objet de poids c-m avec celui de poids c+m, l'objet de poids c-m+1 avec celui de poids c+m-1, etc. L'algorithme ci-dessus commencera par ranger 2 par 2 les objets de poids <W. En effet, le poids total de 3 objets est ≥ (c-m)+(c-m+1)+(c-m+2) = 3c-3(m-1) > 3c-c = 2c = W. L'algorithme utilisera donc m/2 sacs pour les m premiers objets. Les objets restants nécessitent m sacs car ils sont tous de poids >c. L'algorithme ci-dessus utilisera donc en tout A(X)=m/2+m=3m/2 sacs. On déduit :

$$\frac{A(X)}{Opt(X)} = \frac{3}{2}$$

Définition

Soient c et ε deux constantes positives et soit P un problème d'optimisation tel que chaque solution admissible de P a une valeur strictement positive. Notons Opt (I) la valeur optimale d'une instance I de P.

Le problème **c-abs-P** consiste à déterminer, pour chaque instance I de P, une solution de valeur A(I) telle que

$Opt(I) \leq A(I) \leq Opt(I)+c$ si P est un problème de minimisation, ou

$Opt(I)-c \leq A(I) \leq Opt(I)$ si P est un problème de maximisation.

Un algorithme qui produit systématiquement de telles solutions est une **approximation c-absolue** pour P.

Le problème **ε-rel-P** consiste à déterminer, pour chaque instance I de P, une solution de valeur A(I) telle que

$Opt(I) \leq A(I) \leq (1+\varepsilon)Opt(I)$ si P est un problème de minimisation, ou

$(1-\varepsilon)Opt(I) \leq A(I) \leq Opt(I)$ si P est un problème de maximisation.

Un algorithme qui produit systématiquement de telles solutions est une **approximation ε-relative** pour P.

Par exemple, on a vu que l'algorithme *bin-packing_1* est une approximation 1-absolue pour le 1er problème de bin packing dans le cas où k=2. On a également vu que l'algorithme *Sac_À_Dos_2* est une approximation ½-relative pour le problème du sac à dos.

Notons MTSP le problème du voyageur de commerce dans lequel les distances satisfont l'inégalité triangulaire (alors que TSP est le même problème sans la contrainte sur les distances). Ce problème est $\mathcal{NP}$–dur. Le théorème suivant montre qu'il est difficile de développer un algorithme polynomial qui soit une approximation c-absolue pour MTSP.

Théorème 11 MTSP ∝ c-abs-MTSP

Preuve

Soit I une instance de MTSP avec distances entières. Considérons l'instance I' construite en multipliant chaque distance par $\lfloor c \rfloor+1$. On a $Opt(I')=(\lfloor c \rfloor+1)Opt(I)$. Soit A une approximation c-absolue à MTSP. On a $(\lfloor c \rfloor+1)Opt(I)=Opt(I')\leq A(I') \leq Opt(I')+c=(\lfloor c \rfloor+1)Opt(I)+c<(\lfloor c \rfloor+1)(Opt(I)+1)$.

En divisant par $(\lfloor c \rfloor+1)$ on a $Opt(I) \leq \frac{A(I')}{\lfloor c \rfloor+1} < Opt(I)+1$, ce qui implique $Opt(I) = \left\lfloor \frac{A(I')}{\lfloor c \rfloor+1} \right\rfloor$ car Opt(I) est entier.

L'algorithme *Minimum spanning tree* est un algorithme polynomial qui est une approximation 1-relative pour MTSP. Le théorème suivant montre que la situation est différente pour le problème TSP (sans la condition sur les distances). Ce théorème a été démontré dans le chapitre traitant de réductions et transformations.

Théorème HAMD ∝ ε-rel-TSP

Étant donné que HAMD est $\mathcal{NP}$–dur, ce théorème indique qu'il est peu probable qu'il existe un algorithme polynomial qui soit une approximation ε-relatives pour le TSP.

Considérons un tout autre problème. Soit G=(V,E) un graphe dans lequel chaque arête e a un poids c_e. Pour un sous-ensemble $W \subseteq V$, on notera E(W) le sous-ensemble d'arêtes de E ayant leurs deux extrémités dans W.

Définition

MAX-CUT est le problème consistant à déterminer une partition de V en 3 sous-ensembles non vides W_1, W_2 et W_3 qui maximise le poids total des arêtes de $E- E(W_1) \cup E(W_2) \cup E(W_3)$. On cherche donc à maximiser le poids total des arêtes qui relient deux ensembles de la partition.

MIN-CLUSTER est le problème consistant à déterminer une partition de V en 3 sous-ensembles non vides W_1, W_2 et W_3 qui minimise le poids total des arêtes de $E(W_1) \cup E(W_2) \cup E(W_3)$. On cherche donc à minimiser le poids total des arêtes qui relient deux sommets d'un même ensemble de la partition.

Pour un graphe donné G=(V,E), soit MAX(G) la valeur optimale du problème MAX-CUT et soit MIN(G) la valeur optimale de MIN-CLUSTER. Il est clair que MAX(G)+MIN(G) est égal au poids total des arêtes de G. Ces deux problèmes sont donc linéairement équivalents (c'est-à-dire MAX-CUT$=^{\ell}$ MIN-CLUSTER). On va cependant démontrer qu'il existe un algorithme polynomial qui est une approximation 1/3-relative pour MAX-CUT alors que ε-rel-MIN-CLUSTER est $\mathcal{NP}$–dur quel que soit ε.

Étant donné un sous-ensemble W de sommets et un sommet $v \notin W$, notons $c(v,W)$ le coût total des arêtes reliant v aux sommets de W. Considérons l'algorithme suivant qui suppose que $V=\{v_1,\ldots,v_n\}$.

Algorithme Max-Cut

Poser $N_1:=\emptyset$, $N_2:=\emptyset$, $N_3:=\emptyset$, *total* :=0 et *interne* :=0;
Pour i=1 à n **faire**
  Soit $j\in\{1,2,3\}$ tel que $c(v_i,N_j)=\min\{c(v_i,N_1), c(v_i,N_2), c(v_i,N_3)\}$
  Poser *interne*:=*interne* + $c(v_i,N_j)$ et $N_j:=N_j\cup\{v_i\}$;
  Poser *total* :=*total* + $c(v_i,N_1) + c(v_i,N_2) + c(v_i,N_3)$;
Retourner *total*-*interne*;

À la fin de l'algorithme, la variable *total* a une valeur égale au poids total de toutes les arêtes du graphe. De plus, il résulte de l'algorithme que *total* ≥ 3 *interne*. On a évidemment MAX(G) ≤ *total*. On déduit donc (1-1/3)MAX(G) ≤ 2/3 *total* ≤ *total* - *interne*, d'où la Propiété suivante.

Propriété *Max-Cut* est une approximation 1/3-relative pour MAX-CUT

Montrons maintenant que MIN-CLUSTER est bien plus dur à approximer

Théorème 3-COL $\propto$ ε-rel-MIN-CLUSTER

Preuve

Étant donné un graphe G=(V,E), notons H_G le graphe complet ayant V comme ensemble de sommets. Le poids c_e d'une arête e dans H_G est défini comme suit :

$$c_e = \begin{cases} 1 & \text{si } e \notin E \\ \lceil (1+\varepsilon)|V|^2 \rceil & \text{sinon} \end{cases}$$

Supposons qu'il existe une approximation ε-relative à MIN-CLUSTER. Notons $A(H_G)$ la solution résultant de l'application de l'algorithme d'approximation sur H_G. On a donc $MIN(H_G) \leq A(H_G) \leq (1+\varepsilon)MIN(H_G)$.

- Si G est colorable en 3 couleurs, alors il existe une 3-partition $\{N_1,N_2,N_3\}$ de V tel que chaque arête dans $E(N_1)\cup E(N_2)\cup E(N_3)$ est de coût 1. On déduit que $MIN(H_G) < |V|^2$. On a donc $A(H_G) < (1+\varepsilon)|V|^2$.
- Si G n'est pas colorable en 3 couleurs, alors $E(N_1)\cup E(N_2)\cup E(N_3)$ contient au moins une arête de coût $\lceil(1+\varepsilon)|V|^2\rceil$ pour toute 3-partition $\{N_1,N_2,N_3\}$ de V. On a donc $A(H_G)\geq MIN(H_G)\geq\lceil(1+\varepsilon)|V|^2\rceil$.

Pour résoudre 3-COL, il suffit donc de vérifier si $A(H_G) < (1+\varepsilon)|V|^2$.

Définition

Un algorithme A est un **schéma d'approximation** pour un problème P si ∀ε>0 et ∀ instance I de P, cet algorithme permet de déterminer en temps polynomial en |I| une solution de valeur A(I) telle que

$Opt(I) \leq A(I) \leq (1+\varepsilon)Opt(I)$ si P est un problème de minimisation, ou
$(1-\varepsilon)Opt(I) \leq A(I) \leq Opt(I)$ si P est un problème de maximisation.

Un schéma d'approximation A est **complètement polynomial** si le temps requis pour déterminer A(I) est un polynôme en |I| et 1/ε.

Il n'existe probablement aucun schéma d'approximation complètement polynomial pour le problème du bin packing dans lequel on veut minimiser le nombre de sacs. En effet, supposons qu'il existe un tel algorithme A. Étant donnée une instance I à n objets, appliquons cet algorithme avec ε=1/(n+1). On va donc obtenir en temps polynomial une solution de valeur A(I) telle que $Opt(I) \leq A(I) \leq (1+ 1/(n+1))Opt(I) = Opt(I) + Opt(I)/(n+1) < Opt(I) +1$. Comme A(I) est entier, on déduit que Opt(I)=A(I) et on peut donc résoudre le problème de bin packing en temps polynomial, ce qui est peu probable puisqu'il est $\mathcal{NP}$–dur.

Schéma d'approximation polynomial pour le problème du sac à dos

Poser k=min{ $\left\lceil \frac{1}{\varepsilon} \right\rceil - 2, n$ } et best:=0

Pour tout sous-ensemble $K \subseteq \{1,\ldots,n\}$ tel que $|K| \le k$ faire

Si $\sum_{i\in K} w_i \le W$ alors

Appliquer l'algorithme B à l'instance I^K obtenue en ne considérant que les objets O_j tel que $j \notin K$ et $v_j \le \min\{v_r \mid r \in K\}$ et en fixant la capacité du sac à $W - \sum_{i\in K} w_i$.

Soit J^K l'ensemble des indices des objets faisant partie de la solution produite par B.

Si $\sum_{i\in K} v_i + B(I^K) >$ best alors mémoriser $K \cup J^K$ comme meilleure solution et poser best := $\sum_{i\in K} v_i + B(I^K)$

Théorème	L'algorithme ci-dessus est un schéma d'approximation pour le problème du sac à dos

Preuve

Soit I une instance quelconque du problème de sac à dos, et soit s(I) une solution optimale de valeur Opt(I). Si s(I) contient au plus k objets alors cette solution (ou une solution de même valeur) est clairement trouvée par l'algorithme ci-dessus. On peut donc supposer k<n, ce qui signifie que $k = \lceil 1/\varepsilon \rceil - 2$.

Soit K* l'ensemble des indices des k objets de plus grande valeur dans s(I). On a $Opt(I) = \sum_{i\in K^*} v_i + Opt(I^{K^*})$.

Soit H(I) la valeur de la solution produite par l'algorithme ci-dessus. Notons $B(I^{K^*})$ la solution produite par l'algorithme B appliqué à I^{K^*}. On a donc $H(I) \ge \sum_{i\in K^*} v_i + B(I^{K^*}) \ge \sum_{i\in K^*} v_i + 1/2\ Opt(I^{K^*})$.

À titre de rappel, $B(I^{K^*}) = \max\{A(I^{K^*}), v_{max}\}$, où v_{max} est la valeur maximale d'un objet dans I^{K^*}. De plus, on a vu que la valeur optimale du problème I^{K^*} sans contrainte d'intégralité est bornée supérieurement par $A(I^{K^*})+v_{max}$ car une telle solution est obtenue en rajoutant éventuellement une partie d'un objet à la solution produite par l'algorithme A. En résumé, comme I^{K^*} ne contient que des objets de valeur $v_j \le \min\{v_r \mid r \in K^*\}$, on déduit que $Opt(I^{K^*}) \le A(I^{K^*}) + \min\{v_i \mid i \in K^*\}$.

– Si $\sum_{i\in K^*} v_i \ge \frac{k}{k+2} Opt(I)$ alors

$$H(I) \ge \sum_{i\in K^*} v_i + 1/2\ Opt(I^{K^*}) = \sum_{i\in K^*} v_i + 1/2\ (Opt(I) - \sum_{i\in K^*} v_i) = 1/2\ (Opt(I) + \sum_{i\in K^*} v_i)$$

$$\ge 1/2\ (Opt(I) + \frac{k}{k+2} Opt(I)) = \frac{k+1}{k+2} Opt(I)$$

– Si $\sum_{i\in K^*} v_i < \frac{k}{k+2} Opt(I)$ alors

$$Opt(I^{K^*}) \le A(I^{K^*}) + \min\{v_i \mid i \in K^*\} < A(I^{K^*}) + \frac{1}{k+2} Opt(I) \le B(I^{K^*}) + \frac{1}{k+2} Opt(I).$$ On déduit :

$$H(I) \ge \sum_{i\in K^*} v_i + B(I^{K^*}) \ge \sum_{i\in K^*} v_i + Opt(I^{K^*}) - \frac{1}{k+2} Opt(I) = Opt(I) - \frac{1}{k+2} Opt(I) = \frac{k+1}{k+2} Opt(I)$$

Pour chacun des deux cas, on a $H(I) \ge \frac{k+1}{k+2} Opt(I) = (1 - \frac{1}{k+2}) Opt(I) = (1 - \frac{1}{\left\lceil \frac{1}{\varepsilon} \right\rceil}) Opt(I) \ge (1 - \varepsilon)\ Opt\ (I)$.

Étudions maintenant le temps de calcul.

- Le tri des objets peut être fait une fois pour toute, la liste triée étant ensuite utilisée à chaque appel à l'algorithme B. Ceci prend un temps $\in O(n \log_2 n)$
- Chaque appel à l'algorithme B prend un temps $\in O(n)$.

Comme le nombre de sous-ensembles d'au plus k éléments $\in O(n^k)$, on déduit que le temps total d'exécution est $O(n \log_2 n + n^{k+1}) = O(n \log_2 n + n^{\lceil 1/\varepsilon \rceil - 1}) \subseteq O(n \log_2 n + n^{1/\varepsilon})$.

Schéma d'approximation complètement polynomial pour le problème du sac à dos

Considérons le problème du sac à dos avec capacité W et n objets $O_1,\dots,O_n$ de poids w_i et de valeur v_i, Étant donné $k\in\{1,..,n\}$ et un entier positif z, nous allons considérer le problème **Dual_Sac(k,z)** suivant qui consiste à déterminer la plus petite capacité de sac nécessaire pour atteindre une valeur z à l'aide des objets $O_1,\dots,O_k$.

$$\begin{cases} \text{Min} & \sum_{i=1}^{k} x_i w_i \\ \text{s.c.} & \sum_{i=1}^{k} x_i v_i \geq z \\ & 0 \leq x_i \leq 1 \ \text{ entiers} \end{cases}$$

Comme d'habitude, nous noterons Opt(I) la valeur optimale d'une instance I du problème du sac à dos. Nous noterons Dual(I,k,z) la valeur optimale du problème Dual_Sac(k,z). On a donc :

$$\text{Dual}(I,n,z) \leq W \quad \Leftrightarrow \quad \text{Opt}(I) \geq z$$

Résoudre le problème du sac à dos revient donc à déterminer la valeur z maximale telle que Dual(I,n,z)≤W. Ceci peut être réalisé à l'aide d'un algorithme de programmation dynamique. On a

$$\text{Dual}(I,1,z) = \begin{cases} w_1 & \text{si } 0 < z \leq v_1 \\ 0 & \text{si } z = 0 \\ \infty & \text{si } z > v_1 \end{cases}$$

$$\text{Dual}(I,k,z) = \min \{ w_k + \text{Dual}(I,k-1,z-v_k), \text{Dual}(I,k-1,z) \} \quad \text{pour tout } k>1$$

Notons V la valeur totale de tous les objets, et soit U une borne supérieure sur Opt(I). On a donc U≤V. On peut ainsi calculer Dual(I,k,z) pour tout k=1,…,n et pour tout z=0,…,U. On peut donc déterminer Opt(I) en temps $O(nU)\subseteq O(nV)$. Ce n'est pas un algorithme polynomial car V n'est pas polynomial en la taille de l'instance I. (On dit qu'il s'agit d'un algorithme pseudo-polynomial).

Exemple

Pour le problème suivant de sac à dos à 7 objets et avec W=9

i	1	2	3	4	5	6	7
v_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

On obtient le tableau suivant

z \ k	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
3	2	2	2	2	2	2	2
4	2	2	2	2	2	2	2
5	2	2	2	2	2	2	2
6	2	2	2	2	2	2	2
7	∞	5	5	5	5	5	5
8	∞	5	5	5	5	5	5
9	∞	5	5	5	5	5	5
10	∞	5	5	5	5	5	5
11	∞	5	5	5	5	5	5
12	∞	∞	8	8	7	7	7
13	∞	∞	8	8	8	8	8
14	∞	∞	8	8	8	8	8
15	∞	∞	11	9	9	9	**9**
16	∞	∞	11	11	10	10	10

Étant donnée une instance I du problème du sac à dos, et étant donné $s>0$, notons I^s l'instance obtenue en remplaçant chaque v_i par $\lfloor v_i/s \rfloor$. Soit X^* l'ensemble des indices des objets à prendre pour obtenir la solution optimale de I, et soit X^s l'ensemble des indices des objets à prendre pour obtenir la solution optimale de I^s. Notons A(I) la valeur totale des objets d'indices appartenant à X^s. On a

$$A(I) = \sum_{i \in X^S} x_i v_i \geq s \sum_{i \in X^S} x_i \left\lfloor \frac{v_i}{s} \right\rfloor \geq s \sum_{i \in X^*} x_i \left\lfloor \frac{v_i}{s} \right\rfloor.$$

$$> \sum_{i \in X^*} x_i (v_i - s) = Opt(I) - s \sum_{i \in X^*} x_i \geq Opt(I) - sn$$

Si $s \leq \varepsilon\, Opt(I)/n$ alors on a montré que $A(I) > Opt(I) - \varepsilon\, Opt(I) = (1-\varepsilon)Opt(I)$.

On peut par exemple choisir

- $s = \varepsilon V/n^2$ puisque $V/n \leq \max\{v_1,\ldots,v_n) \leq Opt(I)$, ou
- $s = \varepsilon B(I)/n$ où B(I) est la valeur de la solution produite par *Sac_À_Dos_2* (car $B(I) \leq Opt(I)$).

Voici finalement l'algorithme proposé

1. Déterminer B(I) à l'aide de *Sac_À_Dos_2*.
2. Si $\varepsilon < 2n/B(I)$ alors déterminer Opt(I) à l'aide de l'algorithme de programmation dynamique avec 2B(I) comme borne supérieure sur Opt(I). STOP.

   Sinon, poser $s=\varepsilon B(I)/n \geq 2$
3. Construire l'instance I^s en remplaçant chaque v_i par $\lfloor v_i/s \rfloor$
4. Déterminer $Opt(I^s)$ à l'aide de l'algorithme de programmation dynamique avec $\lfloor 2B(I)/s \rfloor$ comme borne supérieure sur $Opt(I^s)$.

Théorème	L'algorithme ci-dessus est un schéma d'approximation complètement polynomial pour le problème du sac à dos

Preuve

À l'étape 2, 2B(I) est bien une borne supérieure sur Opt(I) car on a démontré que $B(I)/Opt(I) \geq ½$.

À l'étape 4, $\lfloor 2B(I)/s \rfloor$ est bien une borne supérieure sur $Opt(I^s)$ car $s\, Opt(I^s) \leq Opt(I) \leq 2B(I)$.

Si $\varepsilon < 2n/B(I)$, l'algorithme ci-dessus détermine l'optimum. Sinon, d'après ce qui précède, nous avons démontré que l'algorithme détermine une solution de valeur $A(I) > (1-\varepsilon)Opt(I)$.

Analysons le temps de calcul :

- L'étape 1 peut être effectuée en $O(n \log_2 n)$
- L'étape 2 nécessite un temps $\in O(nB(I)) \subseteq O(n^2/\varepsilon)$
- L'étape 3 prend un temps $\in O(n)$
- L'étape 4 prend un temps $\in O(n \lfloor 2B(I)/s \rfloor) = O(n \lfloor 2n/\varepsilon \rfloor) \subseteq O(n^2/\varepsilon)$

En résumé, l'algorithme s'exécute en $O(n \log_2 n + n^2/\varepsilon)$. Il s'agit donc bien d'un schéma d'approximation complètement polynomial.

Si on choisit $s = \varepsilon V/n^2$ au lieu de $\varepsilon B(I)/n$ à la fin de l'étape 2, et si on utilise la somme des valeurs des objets comme borne supérieure sur $Opt(I^s)$ lorsqu'on applique l'algorithme de programmation dynamique, alors le temps nécessaire à l'exécution de l'étape 4 $\in O(n \sum_{i=1}^{n} \left\lfloor \frac{v_i}{s} \right\rfloor) \subseteq O(nV/s) = O(n^3/\varepsilon)$.