TD 11 – Approximations et NP-complétude

Exercice 1. *Approximabilité de SUBSET-SUM*

Dans un TD précédent, on s'est intéressé au problème de décision SUBSET-SUM consistant à savoir s'il existe un sous-ensemble d'entiers de S dont la somme vaut exactement t. Le problème d'optimisation qui lui est associé prend aussi en entrée un ensemble d'entiers strictement positifs S et un entier t, il consiste à trouver un sous-ensemble de S dont la somme est la plus grande possible sans dépasser t (cette somme qui doit donc approcher le plus possible t sera appelée *somme optimale*).

On suppose que $S = \{x_1, x_2, \ldots, x_n\}$ et que les ensembles sont manipulés sous forme de listes triées par ordre croissant. Pour une liste d'entiers L et un entier x, on note $L + x$ la liste d'entiers obtenue en ajoutant x à chaque entier de L. Pour les listes L et L', on note **Fusion**(L, L') la liste contenant l'union des éléments des deux listes.

Premier algorithme

Algorithm 1: Somme(S, t)

début
- $n \leftarrow |S|$;
- $L_0 \leftarrow \{0\}$;
- **pour** i **de** *1* **à** n **faire**
  - $L_i \leftarrow$ **Fusion**$(L_{i-1}, L_{i-1} + x_i)$;
  - Supprimer de L_i tout élément supérieur à t;
- **retourner** *le plus grand élément de* L_n;

1. Quelle est la distance entre la valeur retournée par cet algorithme et la somme optimale ?
2. Quelle est la complexité de cet algorithme dans le cas général ? Et si pour un entier $k \geq 1$ fixé, on ne considère que les entrées telles que $t = \mathcal{O}(|S|^k)$, quelle est la complexité de cet algorithme ?

Deuxième algorithme Cet algorithme prend en entrée un paramètre ϵ en plus, où ϵ est un réel vérifiant $0 < \epsilon < 1$.

Algorithm 2: Somme-avec-seuillage(S, t, ϵ)

début
- $n \leftarrow |S|$;
- $L_0 \leftarrow \{0\}$;
- **pour** i **de** *1* **à** n **faire**
  - $L_i \leftarrow$ **Fusion**$(L_{i-1}, L_{i-1} + x_i)$;
  - $L_i \leftarrow$ **Seuiller**$(L_i, \epsilon/2n)$;
  - Supprimer de L_i tout élément supérieur à t;
- **retourner** *le plus grand élément de* L_n;

L'opération **Seuiller** décrite ci-dessous réduit une liste $L = < y_0, y_1, \ldots, y_m >$ (supposée triée par ordre croissant) avec le seuil δ :

Algorithm 3: Seuiller(L, δ)

début
- $m \leftarrow |L|$;
- $L' \leftarrow < y_0 >$;
- $dernier \leftarrow y_0$;
- **pour** i **de** *1* **à** m **faire**
  - **si** $y_i > (1+\delta)dernier$ **alors**
    - Insérer y_i à la fin de L';
    - $dernier \leftarrow y_i$;
- **retourner** L';

3. Évaluer le nombre d'éléments dans L_i à la fin de la boucle. En déduire la complexité totale de l'algorithme. Pour donner la qualité de l'approximation fournie par cet algorithme, borner le ratio valeur retournée/somme optimale.

Exercice 2. *Approximabilité de KNAPSACK*

On s'intéresse au problème du sac-à-dos :
*I*nstance : Un ensemble fini X d'objets; pour chaque objet $x_i \in X$, une valeur $p_i \in \mathbb{N}$ et une taille $a_i \in \mathbb{N}$. Une capacité $B \in \mathbb{N}$.
*S*olution : Un sous-ensemble Y de X tel que $\sum_{x_i \in Y} a_i \leq B$
*M*esure : La valeur totale des objets choisis, i.e. $\sum_{x_i \in Y} p_i$.
On notera $m^*(x)$ la mesure optimale (maximale).

1. Formuler le problème de décision associé et montrer qu'il est $\mathcal{NP}$-complet.
2. Mais alors, comment a-t-on pu résoudre ce problème en cours par programmation dynamique ?

Glouton Dans l'algorithme glouton, on trie les éléments par rapports p_i/a_i décroissants, et on choisit chaque élément examiné dans cet ordre s'il y a la place pour lui. Soit $m_g(x)$ la mesure de l'algorithme glouton.

3. Soit K un entier arbitrairement grand. Construire une instance x telle que $m^*(x)/m_g(x) > K$.
4. Soit p_{max} la valeur maximale d'un objet. Montrer que $m^*(x)/\max\{m_g(x), p_{max}\} < 2$.
   Indication : soit j l'indice du premier élément que l'algorithme glouton ne prend pas ; montrer que $m^*(x) \leq \sum_{i=1}^{j} p_i$.

Programmation dynamique revisited

5. Pour $1 \leq k \leq n$ et $0 \leq p \leq \sum_{i=1}^{n} p_i$, on cherche, parmi les sous-ensembles de $\{x_1, x_2, \ldots, x_k\}$ de valeur totale égale à p et de taille majorée par B, un qui soit de taille minimale. On note $M^*(k,p)$ une solution optimale de ce problème et $S^*(k,p)$ la taille correspondante. Expliquer comment calculer les $M^*(k,p)$ par programmation dynamique. Quelle est la complexité de la résolution du problème du sac-à-dos ?
6. Pour tout rationnel $r > 1$, on considère le schéma d'approximation suivant : soit p_{max} la valeur maximale d'un objet et $t = \lfloor \log(\frac{r-1}{r}\frac{p_{max}}{n}) \rfloor$. On résout par programmation dynamique comme plus haut une instance modifiée du problème original : les tailles des n objets sont toujours a_i mais les valeurs sont $p'_i = \lfloor \frac{p_i}{2^t} \rfloor$. Soit $m_{AS}(x,r)$ la mesure de la solution ainsi obtenue.
   - **(a)** Montrer que la complexité du schéma d'approximation est $O(\frac{r}{r-1}n^3)$.
   - **(b)** Montrer que $m^*(x)/m_{AS}(x,r) \leq r$.
     Indication : montrer que $\frac{m^*(x) - m_{AS}(x,r)}{m^*(x)} \leq \frac{n2^t}{p_{max}}$.