TD 11 – Approximations (corrigé)

(Nuages) **Exercice 1.** *Clustering*

On considère n points dans un espace métrique (les distances entre les points satisfont l'inégalité triangulaire). On veut partitionner les points en k groupes de manière à minimiser le plus grand diamètre d'un groupe. Le diamètre d'un groupe est la distance maximale entre deux points de ce groupe. Noter que n et k sont fixés dans l'énoncé du problème.

1. On suppose que le diamètre optimal d est connu. Trouver une 2-approximation pour le problème.
   ☞ On propose l'algorithme suivant :

Entrées: $\mathcal{S}$ // ensemble de points
Résultat: $\mathcal{P} = \{\emptyset\}$ // partition des points
début
  pour *i allant de* 1 *à k et* $\mathcal{S} \neq \emptyset$ **faire**
    choisir au hasard un point $p \in S$;
    $C = \{p' | d(p, p') \leq d\}$;
    $\mathcal{P} = \mathcal{P} \cup \{C\}$;
    $\mathcal{S} \leftarrow \mathcal{S} \setminus C$;
  retourner $\mathcal{P}$;

L'espace étant métrique, le diametre d'un groupe est au plus $2d$. De plus chaque point est exactement dans un groupe. En effet, par construction chaque point est au plus dans un groupe. De plus si il existe point qui n'est dans aucun groupe, il doit être à plus de d des k points à l'origine des groupes. cela implique que toute k-partition aura un diamètre supérieure à d, ce qui contredit l'optimalité de d.

2. On considère l'algorithme qui, k fois, choisit comme "centre" le point à distance maximale de tous les centres déjà choisis, puis alloue chaque point au centre le plus proche. En faisant le lien avec la question précédente, montrer que cet algorithme est une 2-approximation pour le problème.
   ☞ Nous appellerons A l'algorithme précédent et B l'algorithme proposé dans cette question. À chaque itération l'algorithme A peut choisir soit le même point que B soit un autre point. Comme A est une 2-approximation, si A et B choisissent les même ensemble de points, B sera également une 2-aproximation.
   Lemme 1. *Soit p le centre choisi par l'algorithme B à l'itération i. Soit l'algorithme A peut choisir le point p comme centre, soit l'algorithme B réalise déjà une 2-approximation.*

   Preuve : À une itération donnée, Si A ne peut pas choisir p, cela signifie que p est au plus à une distance d de tous les centre déjà choisi. Mais comme p est choisi par B, cela signifie que tous les points sont au plus à une distance d des centres déjà choisis. Cela implique que d'assigner chaque point au centre le plus proche donne une 2-approximation

(CouvertureSommets) **Exercice 2.** *Vertex Cover*

Nous considérons ici le problème de la couverture par sommets : étant donné un graphe non-orienté $G = (V, E)$ et une fonction de poids sur les sommets $w : V \to \mathbb{Q}+$, trouver $S \subseteq V$ de poids minimum qui couvre les arêtes (c'est à dire $\forall\, e \in E, \exists\, s \in S : s \in e$). En cours, nous avons vu une 2-approximation dans le cas où w est constante. Nous allons étudier ici une 2-approximation dans le cas général. Pour ce faire nous nous intéressons à un type de fonction de poids particulier : les fonctions de poids par degré. w est une fonction de poids par degré si $\exists c > 0, \forall v \in V : w(v) = c \cdot deg(v)$.

1. Soit $\{G = (V, E), w\}$ une instance du problème telle que w est une fonction de poids par degrés. Montrer que $w(V) \leq 2 \cdot OPT$ où $w(V) = \sum_{v \in V} w(v)$.
   ☞ OPT couvre toutes les arêtes, on a donc : $\sum_{v \in OPT} deg(v) \geq m$ et par définition de c il vient : $OPT \geq c \cdot m$. Or $w(v) = c \cdot \sum_{v \in V} deg(v) = 2 \cdot c \cdot m$, d'où $w(v) \leq 2 \cdot OPT$.
   On sait donc traiter les instances avec une fonction de poids par degrés. La méthode du mille-feuille consiste alors à décomposer une instance quelconque en une famille d'instances avec fonction de poids par degrés.

2. À chaque étape de la décomposition, on cherche la plus grande fonction de poids par degrés p inférieure à w. Expliquer comment calculer p.

☞ On cherche une fonction w' telle que :
— $\forall\ v \in V, w'(v) \leq w(v)$, (inférieur à w)
— $\exists\ c', w'(v) = c' \cdot deg(v)$, (par degrés)
— w' est maximale pour les fonctions par degrés. En travaillant dans $\mathbb{R} \cup +\infty$:

$$\forall x \in V, c \leq \frac{w(v)}{deg(v)} \Leftrightarrow c' \leq \min_{v \in V} \frac{w(v)}{deg(v)}$$

On prend alors $c' = \min_{v \in V} \frac{w(v)}{deg(v)}$ car c'est le plus grand que l'on puisse choisir.

L'algorithme du mille-feuille est le suivant :

Données:
$G = (V, E)$;
une fonction de poids quelconque w;
début

1 $t \leftarrow 0$;
2 $G_0 \leftarrow G$;
3 $w_0 \leftarrow w$;
4 **tant que** $G_t = (V_t, E_t)$ *contient une arête* **faire**
5 $D_t \leftarrow \{u \in V_t : deg_t(u) = 0\}$;
6 $p_t \leftarrow$ plus grande fonction de poids par degrés inférieure à w_t dans G_t;
7 $S_t \leftarrow \{u \in V_t : p_t(u) = w_t(u)\}$;
8 $G_{t+1} \leftarrow G_t \setminus (D_t \cup S_t)$;
9 $w_{t+1} \leftarrow w_t - p_t$;
10 $t \leftarrow t + 1$;
11 **retourner** $C = \bigcup_{k=0}^{t-1} S_k$

---

3. Montrer que l'algorithme termine en temps polynomial.

☞ On a à chaque étape $|S_t| \geq 1$, car p_t est maximale et $G_{t+1} \leftarrow G_t\ (D_t \cup S_t)$ donc $|G_{t+1}| \leq |G_t|$; on exécute donc au maximum n fois la boucle **tant que**.
— ligne **??** en $\mathcal{O}(n)$
— ligne **??** en $\mathcal{O}(n)$
— ligne **??** en $\mathcal{O}(n)$
— ligne **??** en $\mathcal{O}(n^2)$
— ligne **??** en $\mathcal{O}(n)$

4. Montrer que l'ensemble C de sommets retournés par l'algorithme est bien une couverture.

☞ Soit $e = (u_l, u_m) \in E$, $\exists\ i$ tel que $e \in G_i$, $e \notin G_{i+1}$. On peut supposer : $u_l \in (S_i \cup D_i)$. On a $deg_i(u_l) \geq 1 \Rightarrow u$ł $\notin D_i$ et donc $u_l \in S_i$. Alors $u_l \in C$et e est couvert par C. donc C est bien une couverture.

5. Pour tout $v \in C$, exprimer $w(v)$ en fonction des poids par degrés p_k. Qu'en est-il pour $v \notin C$? Remarque : on pourra poser $p_k(u) = 0$ pour tout sommet $u \notin G_k$.

☞ À chaque étape, $w_{t+1} \leftarrow w_t - p_t$. Soit $v \in C : v \in S_j$ et on cherche un majorant de $w(v)$. $w(v) = \sum_{k=0}^{j-1} p_k(v) + w(v)$ et $v \in S_j$ donc $w_j(v) = p_j(v)$. D'où $w(v) = \sum_{k=0}^{j} p_k(v)$. Si $v \notin C : v \in D_k$ et on cherche un minorant de $w(v)$.

$$w(v) = \sum_{k=0}^{K-1} p_k(v) + w_k(v) \geq \sum_{k=0}^{K} p_k(v)$$

(car $p_K(v) = 0$)

À partir de maintenant, on notera C^* une couverture par sommets optimale pour l'instance de départ $\{G = (V, E), w\}$.

6. Pour toute étape i de l'algorithme, comparer $p_i(C \cap G_i)$ et $p_i(C^* \cap G_i)$.
Indication : on remarquera que $C \cap G_i$ et $C^* \cap G_i$ sont deux couvertures par sommets de G_i.
☞

Une couverture est stable par restriction à un sous-graphe. Soit OPT une couverture optimale de (G_i, p_i). D'apres la question 1, $p_i(C \cap G_i) \leq 2 \cdot OPT$. Comme $C^* \cap G_i$ est une couverture de G_i , $p_i(C^* \cap G_i) \geq OPT$. Donc $p_i(C \cap G_i) \leq 2 \cdot p_i(C^* \cap G_i)$.

7. Montrer que l'algorithme est une 2-approximation du problème de la couverture par sommet minimum avec des poids arbitraires. Trouver un exemple où l'algorithme renvoie effectivement une solution de poids $2.OPT$.
☞

(SubsetSum) **Exercice 3.** *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 ?
☞ On a un ensemble $S = \{x_1, \ldots, x_n\}$ d'entiers strictement positifs ainsi qu'un entier t. On cherche une somme partielle d'éléments de S maximale et inférieure à t. On appellera somme optimale ce nombre.
Dans le premier algorithme, L_i représente l'ensemble des sommes partielles des éléments de $\{x_1, \ldots, x_i\}$ inférieures à t. L_n représente donc l'ensemble des sommes partielles de S inférieures à t. Par conséquent, $\max(L_n)$ renvoie exactement la somme optimale de S. De plus, cet algorithme a testé toutes les sommes partielles inférieures à t.

La différence entre le résultat trouvé et la somme optimale est 0.

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 ?
☞
— Dans le pire des cas, on teste toutes les sommes partielles de S soit une complexité en $O(2^n)$: il existe des entrées telles que $|L_i| = 2^i$, prendre par exemple $S = \{1, 2, 2^2, \ldots, 2^i, \ldots, 2^{n-1}\}$ et t grand ($\geq 2^n$). Or comme la fusion à chaque étape est en $O(|L_{i-1}|)$, la complexité totale est exponentielle en n.
— Si $t = O(|S|^k) = O(n^k)$ pour k un entier fixé, alors $|L_i| = O(|S|^k)$. Au pas i de la boucle, la fusion et la suppression se font en $O(|L_i|)$. Comme il y a n pas dans la boucle, la complexité totale est $O(|S|^{k+1})$.

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 ← |L|;
    L' ←< y_0 >;
    dernier ← y_0;
    pour i de 1 à m faire
        si y_i > (1 + δ)dernier alors
            Insérer y_i à la fin de L';
            dernier ← 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.

☞ Considérons une liste l = $\{y_0, \ldots, y_m\}$ d'entiers triés par ordre croissant et $\delta > 0$. L'opération de seuillage va consister à dire : soit y' le dernier élément gardé de l (on garde y_0 au départ) si $y_i > y' * (1 + \delta)$ alors on va garder y_i. En gros, on enlève les éléments qui sont trop proches d'autres éléments.

L'algorithme proposé va fonctionner de la même façon que le précédent sauf qu'il va seuiller à chaque pas de la boucle L_i par rapport à $\epsilon/2n$. On a alors $L_n = \{y_0, \ldots, y_m\}$. On va essayer de majorer m. On sait que :

$t \geq y_m \geq (1 + \epsilon/2n)^{m-1} * y_0 \geq (1 + \epsilon/2n)^{m-1}$

D'où : $m \leq \ln t / \ln(1 + \epsilon/2n) + 1 \leq \frac{2n \log(t)}{\epsilon \log(2)} + 1$, car $(m-1)\log(1 + \epsilon/2n) \leq \log(t)$ et donc $(m-1)\epsilon/2n \log(2) \leq \log(t)$ (car on a l'inégalité $x \log(2) \leq log(1 + x)$ quand $0 \leq x \leq 1$).

m est donc polynomial en $\ln t$, en $1/\epsilon$ et en n, donc polynomial en la taille des données. L'algorithme est en $O(mn)$, donc polynomial en la taille des données. Complexité totale : $O(n \times \left(\frac{2n \log(t)}{\epsilon \log(2)} + 1\right)) = O(\frac{n^2 \log(t)}{\epsilon})$.

Il faut maintenant vérifier qu'on a bien trouvé une $(1 + \epsilon)$Approximation en montrant que : $max\{L_n\} \geq (1 + \epsilon) * Opt$(somme optimale)

On note P_i l'ensemble des sommes partielles de $\{x_1, \ldots, x_i\}$ inférieures à t.

Invariant : $\forall x \in P_i, \exists y \in L_i,\ x/(1 + \delta)^i \leq y \leq x$

Par récurrence :

— $i = 0$: OK

— On suppose que c'est vrai pour $(i-1)$ et on le montre pour i : Soit $x \in P_i$
$P_i = P_{i-1} \cup (P_{i-1} + x_i)$
$x = x' + e$ avec $x' \in P_{i-1},\ e = 0\ ou\ e = x_i$
$x' \in P_{i-1}$ donc $\exists y',\ x'/(1 + \delta)^i \leq y' \leq x'$
Si y' + e est conservé par le seuillage :
$(x' + e)/(1 + \delta)^i \leq x'/(1 + \delta)^{i-1} + e \leq y' + e \leq x' + e = x$
Si y' + e n'est pas conservé par le seuillage :
$\exists y'' \in L_i,\ y'' \leq y' + x_i \leq y'' * (1 + \delta)$
$(y' + e)/(1 + \delta) \leq y'' \leq y' + e \leq x' + e \leq x$
$(x'/(1 + \delta)^{i-1} + e)/(1 + \delta) \leq y''$
$(x' + e)/(1 + \delta)^i \leq y''$
C'est l'encadrement recherché.
Grâce à l'invariant, on obtient : $Algo \geq Opt/(1 + \epsilon/2n)^n \geq Opt(1 + \epsilon)$
On a donc une $(1 + \epsilon)$approximation de Subset-sum polynomiale en la taille des données et polynomiale en $1/\epsilon$.