Devoir Non Surveillé 3

À rendre le 24 février 2025

INFORMATIQUE MPI/MPI*

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 3 parties indépendantes, utilisant de la programmation en langage OCaml, et pas de programmation en langage C.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

— La partie I étudie le problème des k-moyennes et la méthode vue en classe d'un point de vue plus théorique.
— La partie II s'intéresse à la complexité amortie de la structure de donnée unir et trouver implémentée avec union par rang et compression de chemin.
— La partie III s'intéresse à la structure de tas binomiaux et leur implémentation en OCaml.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

Partie I. Problème des k-moyennes

Dans cette partie, on s'intéresse au problème d'optimisation des k-moyennes défini ainsi :

k-means problem

Instance : un ensemble de points $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, avec $n \in \mathbb{N}$, un entier $k \in \mathbb{N}^*$.

Solution : une partition $S = (S_1, \ldots, S_k)$ de X.

Optimisation : minimiser $\displaystyle\sum_{i=1}^{k} \sum_{x_j \in S_i} \|\mathbf{x}_j - \mu_i\|_2^2$, où μ_i est l'isobarycentre des points dans S_i.

Nous rappelons aussi l'algorithme de la méthode des k-moyennes vu en cours :

k-means algorithm

Choisir k points aléatoires $(\mu_1, \ldots, \mu_k)$ parmi X.
Calculer la partition : $S_i = \left\{ \mathbf{x}_j \;\middle|\; i = \operatorname{argmin}_{i' \in [\![1,k]\!]} \|\mathbf{x}_j - \mu'_i\| \right\}$.
Calculer les isobarycentres $G_i = \frac{1}{|S_i|} \displaystyle\sum_{\mathbf{x}_j \in S_i} \mathbf{x}_j$.
Tant qu'il existe j tel que $\mu_j \neq G_j$:
poser $\mu_i = G_i$ pour tout $i \in [\![1, k]\!]$ et recalculer les S_i puis G_i.

Dans un premier temps, on considère l'exemple constitué des 4 points suivants, pour $k = 2$:

$$\mathrm{A}(0,0) \qquad \mathrm{B}(0,2) \qquad \mathrm{C}(10,0) \qquad \mathrm{D}(10,2)$$

☐ 1 – Appliquer la méthode des k-moyennes pour trouver une partition, avec pour initialisation $\mu_1 = \mathrm{A}$ et $\mu_2 = \mathrm{B}$.

☐ 2 – Donner une solution au problème des k-moyennes sur cet exemple. Quelle initialisation permettrait de trouver cette partition à l'aide de la méthode des k-moyennes ?

☐ 3 – Montrer que l'isobarycentre $\mu = \frac{1}{|S|} \displaystyle\sum_{\mathbf{x} \in S} \mathbf{x}$ d'un ensemble de point S vérifie

$$\sum_{\mathbf{x} \in S} (\mathbf{x} - \mu) = \mathbf{0}.$$

ou montrer que pour tout point M de coordonnées $\mathbf{x}_\mathrm{M}$, il vérifie :

$$\sum_{\mathbf{x} \in S} (\mathbf{x} - \mathbf{x}_\mathrm{M}) = |S| \, (\mu - \mathbf{x}_\mathrm{M})$$

En déduire que c'est le point qui minimise la somme des distances au carré :

$$\mu = \operatorname{argmin}_{\mathbf{y}} \sum_{\mathbf{x} \in S} \|\mathbf{x} - \mathbf{y}\|_2^2$$

☐ 4 – Montrer qu'à chaque étape de l'algorithme, la quantité $\displaystyle\sum_{i=1}^{k} \sum_{x_j \in S_i} \|\mathbf{x}_j - \mu_i\|_2^2$ décroît strictement, ou alors l'algorithme s'arête.

☐ 5 – Montrer que l'algorithme termine nécessairement, et donner une borne supérieure sur sa complexité dans le pire des cas.

On considère maintenant le problème de seuil associé au problème des k-moyennes :

k-means problem - seuil

Instance : un ensemble de points $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, avec $n \in \mathbb{N}$, un entier $k \in \mathbb{N}^*$, et $L \in \mathbb{R}$.

Solution : V s'il existe une partition $S = (S_1, \ldots, S_k)$ de X telle que $\sum_{i=1}^{k} \sum_{x_j \in S_i} \|\mathbf{x}_j - \mu_i\|_2^2 \leq L$, où μ_i est l'isobarycentre des points dans S_i. F sinon.

☐ 6 – Montrer que ce problème est dans la classe NP.

On cherche maintenant à montrer que ce problème est NP-complet [1]. On part de la variante du problème 3-Sat suivante, que l'on admettra être NP-difficile :

3-Sat

Instance : Une formule booléenne sous forme normale conjonctive $\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{m} C_j$, où chaque clause $C_j = \ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ possède exactement 3 littéraux $\ell_{j,k} = x_i$ ou $\ell_{j,k} = \bar{x}_i$ pour un certain $i \in [\![1, n]\!]$, et chaque pair de variables x_i, x_j apparaît dans au plus 2 clauses : une fois en tant que $\{x_i, x_j\}$ ou $\{\bar{x}_i, \bar{x}_j\}$, et une fois en tant que $\{\bar{x}_i, x_j\}$ ou $\{x_i, \bar{x}_j\}$.

Solution : V si φ est satisfiable. F sinon.

On introduit aussi le problème Not-all-equal 3Sat dont on admet qu'il se réduit à une généralisation du problème des 2-moyennes.

NaeSat*

Instance : Une formule booléenne sous forme normale conjonctive $\varphi(x_1, \ldots, x_n) = \bigwedge_{j=1}^{m} C_j$, où chaque clause $C_j = \ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ possède exactement 3 littéraux $\ell_{j,k} = x_i$ ou $\ell_{j,k} = \bar{x}_i$ pour un certain $i \in [\![1, n]\!]$, et chaque variable x_i apparaît au moins 2 fois.

Solution : V si il existe un modèle de φ pour lequel chaque clause contient exactement 1 ou 2 littéraux satisfaits. F sinon.

☐ 7 – Soit $\varphi(x_1, \ldots, x_n)$ une instance de 3-Sat. Construire une formule φ' sous forme normale conjonctive équisatisfiable avec φ mais dont chaque variable apparaît exactement 3 fois : 1 fois dans une clause de 3 littéraux, et 2 fois dans des clauses de 2 littéraux.
Indication : On pourra remarquer que $\bar{x}_i \vee x'_i$ et $\bar{x}'_i \vee x_i$ assurent que x_i et x'_i ont même valeur de vérité, et que cette construction peut être généralisé à plus de 2 variables.

On construit à présent une formule φ'' à partir de φ'. Soit m le nombre de clauses de φ' contenant 3 littéraux et m' le nombre de clauses de φ' contenant 2 littéraux. On définit $2m+m'+1$ variables : $s_1, \ldots, s_m$ et $f_1, \ldots f_{m+m'}$ et f, et on définit φ'' de la manière suivante :
La j-ième clause de φ' contenant 3 littéraux $\alpha \vee \beta \vee \gamma$ est remplacé dans φ'' par 2 clauses : $\alpha \vee \beta \vee s_j$ et $\bar{s}_j \vee \gamma \vee f_j$.
La j-ième clause de φ' contenant 2 littéraux $\alpha \vee \beta$ est remplacé dans φ'' par la clause $\alpha \vee \beta \vee f_{m+j}$.
On ajoute à φ'' les $m + m'$ clauses suivantes : $\bar{f}_1 \vee f_2 \vee f, \quad \bar{f}_2 \vee f_3 \vee f, \quad \ldots, \quad \bar{f}_{m+m'} \vee f_1 \vee f$.

☐ 8 – Vérifier que φ'' est bien une instance de NaeSat*.

☐ 9 – Montrer que si v est un modèle de φ'' pour lequel chaque clause contient exactement 1 ou 2 littéraux satisfaits, alors $v(f) = v(f_1) = \ldots = v(f_{m+m'})$.

☐ 10 – Démontrer que $\varphi \mapsto \varphi''$ est une réduction polynomiale de 3-Sat à NaeSat*.

On peut en déduire que le problème de seuil associé au problème d'optimisation des k-moyennes est NP-complet, même si $k = 2$. On peut aussi montrer que le problème des k-moyennes est NP-complet dans un espace euclidien de dimension 2.

1. Pour plus de détails, suivre : The hardness of k-means clustering

Partie II. Unir & trouver

On considère dans cette partie la structure Unir & trouver implémentée par une forêt, avec union par rang et compression de chemin, et on s'intéresse à la complexité amortie de m opérations, chacune `unir` ou `trouver`, dans une ensemble de n éléments. On rappelle une implémentation en langage C.

Union-find structure **C**

```c
struct subset {
  int parent;  // recherche par compression de chemin pour Trouver
  int rang;    // Union par rang .
};
typedef struct subset subset;

subset Make(){ // creer un singleton
  subset sg;
  sg.parent = -1;
  sg.rang = 0;
  return sg;
}

subset *Init(int n){ // creer n singletons dans un tableau
  subset *s = malloc(n * sizeof(subset));
  for (int i = 0; i < n; i = i + 1){
    s[i] = Make();
  }
  return s;
}

int Trouver(int i, subset *s){
  int r = i;
  while (s[r].parent != -1){
    r = s[r].parent;
  }
  int tmp = s[i].parent;
  while (tmp != r){
    s[i].parent = r;
    i = tmp;
    tmp = s[i].parent;
  }
  return r;
}

void Unir(int i, int j, subset *s){
  assert(s[i].parent == -1 && s[j].parent == -1);
  if (i == j){ return; }
  if (s[i].rang == s[j].rang){
    s[j].parent = i;
    s[i].rang = s[i].rang + 1;
  }
  else if (s[i].rang < s[j].rang){
    s[i].parent = j;
  }
  else {
    s[j].parent = i;
  }
}
```

On définit la fonction $\log_2^* : n \mapsto \begin{cases} 0 \text{ si } n \leq 1 \\ 1 \text{ si } n = 2 \\ 1 + \log_2^*(\log_2(n)) \text{ si } n > 2 \end{cases}$.
Elle correspond au nombre de fois qu'il faut appliquer la fonction $\log_2$ à un nombre n pour obtenir un nombre inférieur ou égal à 1.
On définit aussi les intervalles $I_k = \{n \in \mathbb{N} \mid \log_2^*(n) = k\}$.

☐ 11 – Donner les plus petits nombres n tels que $\log_2^*(n)$ vaut : $2, 3, 4, 5, 6$ et montrer que I_k est de la forme $[\![B, 2^B - 1]\!]$.

☐ 12 – Montrer que si s[i]. parent vaut $j \neq -1$, alors s[i]. rang $<$ s[j]. rang.

☐ 13 – Montrer que tout élément i tel que s[i]. parent vaut -1 et s[i]. rang vaut k a au moins 2^k éléments dans l'arbre dont il est racine (c'est-à-dire qu'il existe au moins 2^k éléments j tels que Trouver(j, s) renvoie i).

☐ 14 – En déduire que si le nombre total d'éléments est n, tous les rangs sont inférieurs à $\log_2(n)$.

☐ 15 – Montrer que pour tout k, le nombre de racines de rang k est au plus $\frac{n}{2^k}$.

☐ 16 – Montrer que pour tout $k > \log_2^*(n)$, aucune racine n'a son rang dans I_k.

On définit à présent les 3 quantités suivantes, qui dénombre des liens suivis par les différentes opérations Trouver :
- T_1 le nombre de liens $i \to j$ traversés, avec j une racine (s[j]. parent vaut -1) ;
- T_2 le nombre de liens $i \to j$ traversés, avec i et j qui ont leur rang dans des intervalles I_k différents ;
- T_3 le nombre de liens $i \to j$ traversés, avec i et j qui ont leur rang dans un même intervalle I_k.

☐ 17 – Montrer que $T_1 = O(m)$. On rappelle que m est le nombre total d'opérations Unir et Trouver.

☐ 18 – Montrer que $T_2 = O(m \log_2^*(n))$.

☐ 19 – Pour un i donné dont le rang est dans l'intervalle $[\![B, 2^B - 1]\!]$, montrer que le nombre de liens $i \to j$ traversés, où j n'est pas une racine et a son rang dans $[\![B, 2^B - 1]\!]$, est inférieur à 2^B.

☐ 20 – En déduire une borne supérieure sur T_3.

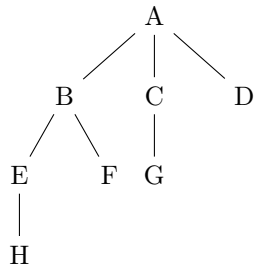
On en déduit la complexité amortie $O(\log_2^*(n))$ par opération Unir ou Trouver.
Avec une analyse encore plus approfondie, on peut montrer la complexité amortie $O(\alpha(n))$, où α est l'inverse de la fonction d'Ackermann (c'est plutôt une réciproque), qui croît encore plus lentement que $\log_2^*$.

Partie III. Tas binomiaux

Arbres binomiaux

Les arbres binomiaux sont une famille d'arbres d'arité quelconque définie par induction :

— Un arbre binomial d'ordre 0 est réduit à sa racine ;
— Un arbre binomial d'ordre k est constitué d'une racine et de k sous-arbres qui sont des arbres binomiaux d'ordres respectifs $k-1, k-2, \ldots, 0$, où les ordres des sous-arbres sont décroissant de gauche à droite.



Exemple d'arbre binomial d'ordre 3

☐ 21 – Montrer que l'on peut constuire un arbre binomial d'ordre k à partir d'arbres binomiaux d'ordre $k-1$.

☐ 22 – Donner le nombre de nœuds dans un arbre binomial d'ordre k. Justifier.

On utilise les types suivants pour représenter les arbres binomiaux en OCaml :

```
(* Les arbres sont de la forme Node (ordre, racine, fils) *)
type 'a arbre = Node of int * 'a * 'a foret
and 'a foret = 'a arbre list
```

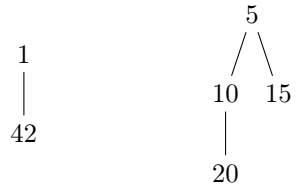
☐ 23 – Écrire une fonction union : 'a arbre –> 'a arbre –> 'a arbre prenant en entrée deux arbres binomiaux d'ordre k et renvoyant un arbre binomial d'ordre $k+1$ contenant les mêmes éléments, avec pour racine la plus petite des deux racines des arbres d'entrée.

☐ 24 – Écrire une fonction est_binomial : int –> 'a arbre –> bool qui teste si un arbre est bien un arbre binomial d'ordre k.

☐ 25 – Calculer la complexité de cette fonction dans le cas ou l'entrée est bien un arbre binomial.

Implémentation des tas binomiaux

Un tas binomial est une forêt d'arbres binomiaux d'ordres distincts, dont chacun des arbres vérifie la propriété de tas : tout nœud qui n'est pas une racine possède une étiquette plus grande que celle de son père. On les représentera en OCaml par le type 'a foret défini précédemment, les arbres étant triés par ordre croissant : les arbres de plus petit ordre sont à gauche de la liste.



Exemple de tas binomial contenant 6 éléments

☐ 26 – Montrer qu'un tas binomial contenant n éléments possède au plus $O(\log n)$ arbres.

☐ 27 – Implémenter l'ajout d'un arbre binomial dans un tas binomial et en déduire l'ajout d'un élément dans un tas binomial.

☐ 28 – Calculer la complexité de cette fonction.

☐ 29 – Proposer un algorithme pour faire l'union de deux tas binomiaux et en déduire un algorithme pour extraire le minimum d'un tas binomial. (on ne demande pas de code pour cette question)

☐ 30 – Calculer la complexité de ces opérations.

☐ 31 – Quelle est la structure de donnée abstraite implémentée par les tas binomiaux ?

Fin de l'épreuve