Contrôle écrit avec **corrigé** INF442
— Traitement des données massives —

École Polytechnique, Département informatique

Frank Nielsen

Promotion X2014
Mercredi 8 juin 2016
14h-17h

Informations générales

— Les exercices sont tous *indépendants* les uns des autres, et peuvent donc être traités dans n'importe quel ordre sur la copie.
— On appreciera la *clarté* (en premier lieu) et la *concision* des réponses (en second lieu). Tout document du cours INF442 autorisé (polycopié, memento C++ et planches des cours en amphithéâtre).
— Tous les problèmes sont structurés en plusieurs sous-problèmes. Indiquez précisément la numérotation des sous-problèmes que vous traitez sur vos feuilles d'examen.
— Prenez 10 minutes de votre temps pour bien lire les énoncés avant de commencer, et gardez au moins 5 à 10 minutes à la fin pour la relecture finale.

Barème

Les problèmes ci-dessous comptent pour la note finale de la façon suivante :

Problème	%
1 — Sur la loi d'Amdahl	10%
2 — Sur l'hypercube	20%
3 — Sur l'arbre recouvrant de poids minimal	40%
4 — Sur quelques extensions des k-moyennes	30%

Solution

I-1. Sur l'hypercube H_d de dimension d, on a $M = 2^d$ nœuds, et donc $\text{speedup}(H_d) = \frac{1}{\alpha_{\text{seq}} + \frac{\alpha_{\text{par}}}{2^d}}$.

I-2. I-2-a. Nous avons $\alpha_{\text{par}} = \frac{6}{10} = \frac{3}{5}$ et $\alpha_{\text{seq}} = 1 - \alpha_{\text{par}} = \frac{2}{5}$. On applique la loi d'Amdahl : $\text{speedup}(M) = \frac{1}{\alpha_{\text{seq}} + \frac{\alpha_{\text{par}}}{M}}$, et on trouve $\text{speedup}(3) = \frac{5}{3} \simeq 1,67 \leq 3$.

I-2-b. L'accéleration asymptotique pour $M \to \infty$ est $\frac{1}{\alpha_{\text{seq}}} = 5/2 = 2,5$.

I-2-c. L'efficacité asymptotique est $\frac{\text{speedup}(M)}{M} \sim \frac{1}{\alpha_{\text{seq}} M} \to 0$ quand $M \to +\infty$.

I-3. On a $\text{speedup}(M) = \frac{1}{\alpha_{\text{seq}} + \frac{\alpha_{\text{par}}}{M}} = \frac{t}{\beta t} = \frac{1}{\beta}$ et $\alpha_{\text{par}} = 1 - \alpha_{\text{seq}}$. On trouve donc $\alpha_{\text{par}} = \frac{M(1-\beta)}{M-1}$.

Pour $\beta = \frac{1}{M}$, on a $\alpha_{\text{par}} = 1$, c'est-à-dire que le programme est 100% parallélisable. Puisque $\text{speedup}(M) = \frac{1}{\beta} \leq M$ (M est l'accéleration optimale), on en déduit que $\beta \geq \frac{1}{M}$.

I-4. I-4-a. On a par définition $\sum_{i=1}^{B} s_i = 1$, et

$$\sum_{i=1}^{B} s_i \times l_i \leq \bar{\alpha}_{\text{par}} = \sum_{i=1}^{B} s_i \times \alpha_{\text{par},i} \leq \sum_{i=1}^{B} s_i \times u_i$$

I-4-b. Soit $P = P_1; P_2$ avec $s_1 = \frac{1}{3}, l_1 = u_1 = \alpha_{\text{par},1} = \frac{1}{2}$ et $s_2 = \frac{2}{3}, l_2 = u_2 = \alpha_{\text{par},2} = \frac{3}{4}$. On a $\bar{\alpha}_{\text{par}} = \frac{1}{2} \times \frac{1}{3} + \frac{2}{3} \times \frac{3}{4} = \frac{2}{3}$. Ainsi, on a $\text{speedup}(M) = \frac{3}{1+\frac{2}{M}}$ et pour $M = 2$, on trouve $\text{speedup}(2) = \frac{3}{2}$.

Solution

II-1. La distance de Hamming entre v et v' avec $\text{Gray}(v) = 0110$ et $\text{Gray}(v') = 1101$ est $\text{XOR}(0110, 1101) = 3$ (nombre de bits différents).

II-2. Puisque un nœud à distance i de v à i bits différents dans le code de Gray, on en déduit qu'il y a $n_i = \binom{d}{i} = \frac{d!}{i!(d-i)!}$ nœuds à distance i de v.

On vérifie ainsi que le nombre de nœuds de l'hypercube est $\sum_{i=0}^{d} n_i = \sum_{i=0}^{d} \binom{d}{i} = 2^d$ en additionnant tous les nombres de nœuds à distance i pour $i \in \{0, ..., d\}$.

II-3. II-3-a. Soit v tel que $\text{dec}(v) = 25$. En représentation binaire, on utilise 5 bits pour les 32 nœuds de H_5, et on a $\text{bin}(v) = 11001$.

II-3-b. On convertit en code de Gray pour obtenir $\text{Gray}(v) = 10101$.

II-3-c. Soit v' tel que $\text{Gray}(v') = 01101$, on a $\text{bin}(v') = 01001$.

II-3-d. On trouve $\text{dec}(v') = 9$.

II-4. Il n'y a pas de cycles de longueur impaire dans un hypercube puisque pour chaque arête traversée on flippe exactement un bit de la représentation du code de Gray entre les nœuds de cette arête. La longueur d'un cycle est donc le nombre de bits flippés dans la représentation du code de Gray des nœuds du cycle. Pour n'importe quel nœud du cycle, en parcourant tous les nœuds du cycle et en revenant à ce nœud initial, on doit donc nécessairement flipper un nombre pair de bits dans la représentation du code de Gray de v. Bien entendu, notons que tous les chemins de longueur paire ne sont pas forcément des cycles !

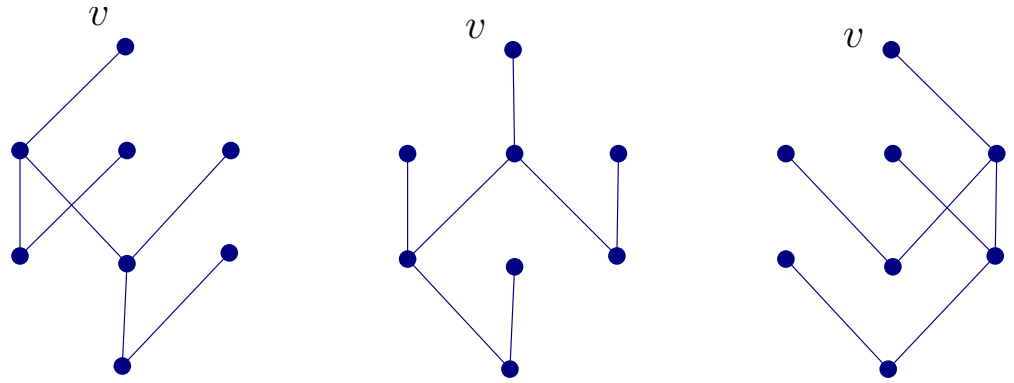
II-5. On décrit l'algorithme de routage `E-cube` comme suit : le nœud v contenant le message l'envoie sur le lien `k=LS1bit(XOR(v,g2))`, et son voisin qui a le code `XOR(v,2**k)` le réceptionne (on flippe le k-ième bit). On répète la procédure jusqu'à temps que `XOR(v,g2)=0`. Il y a l étapes et pour chaque étape, on doit synchroniser tous les processus avec une barrière de synchronisation.

Pour deux nœuds v et v' avec $D_H(v, v') = l$, la complexité du temps de communication est

$$O\left(l(\alpha + \tau L)\right)$$

On pourrait améliorer cette complexité en faisant des communications pipelinées en partitionnant le message en r morceaux.

II-6. Une figure illustrant les trois arbres recouvrants de H_3 partant d'un nœud donné :



Pour deux nœuds v et v' avec $D_H(v, v') = l$, la complexité du temps de communication en utilisant les $\frac{d}{2}$ arbres recouvrants (en fractionnant le message en $\frac{d}{2}$ messages de longueur $\frac{2L}{d}$) est

$$O\left(\left(\tau \frac{2L}{d}\right)\right)$$

L'algorithme Ecube en pseudo-code MPI :

```
Ecube(b1,b2,T)

g1=gray(b1);
g2=gray(b2);

v=g1;
r=XOR(v,g2);

while (r != 0 )
{
k=LS1bit(r);

if (rank == v)
    send(T,k);
else
if (rank == XOR(v,2**k))
receive(T,k);

Barrier();

v=XOR(v,2**k); // 2**k veut dire deux puissance k
r=XOR(v,g2);
}
```

Solution

III-1. T^* est représenté sur la Figure 2 avec son poids total $w(T^*) = 119$.

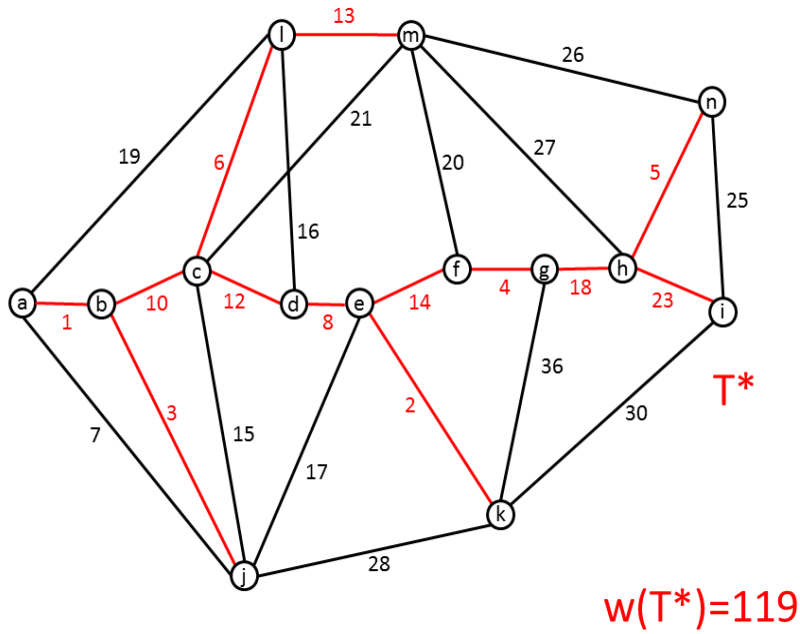


FIGURE 2 – Graphe G et arbre recouvrant minimal

III-2. Algorithme 2 est un pseudo-code de l'algorithme de Kruskal en séquentiel (approche simple)
Algorithme 4 est un pseudo-code de l'algorithme de Kruskal en séquentiel qui utilise la structure *find-union*.

Algorithme 3 est un pseudo-code de l'algorithme de Kruskal en parallèle. L'idée est de partitionner l'ensemble V sur les p processeurs ($V = \cup_{i=1}^{p} V_i$) de sorte que chaque processeur i traite $\frac{n}{p}$ sommets consécutifs de V_i et leur arêtes (ajuster le dernier processeur). Ensuite, chaque processeur détermine son MSF local en utilisant par exemple Algorithme 4 (Kruskal en séquentiel). Enfin, on fait des fusions deux à deux pour unir des MSFs locaux de deux processeurs jusqu'à ce qu'il ne reste qu'un seul processeur qui en 'mergeant' les deux dernier MSFs donne le MST global. Pour merger deux MSFs, on peut également utiliser Algorithme 4 (Kruskal en séquentiel).

Analyse de la complexité :

— Calculer le MSF en local prend $O(\frac{n^2}{p})$. On a au total $\log p$ opérations 'merge' chacune prenant $O(n^2 \log p)$. Pendant chaque 'merge', au plus $O(n)$ arêtes sont envoyées. Ainsi, le temps total en parallèle est $t_p = O(n^2/p) + O(n^2 \log p)$.

— Calcul de l'accélération : $speedup(p) = \frac{t_{seq}}{t_p}$, avec $t_{seq} = O(m + n \log n)$

— Calcul de l'efficacité : $e(p) = \frac{speedup(p)}{p} = O(\frac{m+n \log n}{p})$

III-3. L'Algorithme 1 est très similaire à l'algorithme de Kruskal. L'utilisation du MST dans le single linkage hierarchical clustering peut se faire de la manière suivante : Nous trions les arêtes du MST par ordre croissant de leur poids, et initialisons chaque cluster à un sommet du MST. À chaque étape k du single linkage hierarchical clustering, la k ème arête (x^*, y^*) du MST prise selon l'ordre considéré est celle correspondante à (C_i^*, C_j^*), çest-à-dire $w((x^*, y^*)) = \min_{C_i \in C, C_j \in C} (\min_{x \in C_i, y \in C_j} d(x, y))$.

Ainsi à partir du MST nous pouvons donc reconstruire le dendrogramme T du single linkage hierarchical clustering.

III-4. — Faire une Depth First Search en parallèle pour $C(e), e \notin T^*$.
— Pour $C^{-1}(e), e = (u, v) \in T^*$, faire T_u et T_v de deux couleurs. Tester les arêtes de $e' \in E \backslash T^*$ (peut se faire en parallèle). Si les deux extrémité de e' ont des couleurs différentes alors e' est une arête de remplacement.

III-5. L'arête de remplacement $r(e)$ est la plus petite arête de $C^{-1}(e)$. Il suffit d'appeler l'algorithme précédent pour déterminer $C^{-1}(e)$ et de prendre l'arête $e' = \arg \min_{f \in C^{-1}(e)} w(f)$.

III-6. Les arêtes de remplacement des arêtes de T^* sont données dans le Tableau 1.

e	(a,b)	(b,c)	(c,d)	(d,e)	(e,f)	(f,g)	(g,h)	(h,i)	(b,j)	(e,k)	(c,l)	(i,m)	(h,n)
r(e)	(a,j)	(c,j)	(l,d)	(e,j)	(m,f)	(m,n)	(m,n)	(i,n)	(a,j)	(j,k)	(d,l)	(f,m)	(i,n)
Δw	6	5	4	9	6	22	8	2	4	26	10	7	20

Tableau 1 – Arêtes de remplacement des arêtes de T^*

L'arête la plus vitale au regard du MST est alors (e, k) dont la suppression engendre une augmentation du poids du MST de 26.

C'est l'arête e^* telle que $w(r(e^*)) - w(e^*) = \max_{e \in T^*} w(r(e)) - w(e)$. Calculer donc $r(e)$ pour tout $e \in T^*$ et ensuite déterminer e^* parmi ces arêtes là.

Plus de détails dans le papier [1] (1991).

Entrée : $G = (V, E, w)$ un graphe non orienté connexe pondéré
Sortie : le MST T
Début
$T = \emptyset$;
Trier les arêtes de E par ordre croissant des poids ;
for each $e = (u, v) \in E$ pris dans cet ordre **do**
  if l'ajout de e à T ne cre pas de cycle **then**
    $T = T \cup \{e\}$;
  end if
end for
return T ;

Algorithm 2: Algorithme séquentiel de Kruskal - Approche 1

Entrée : $G = (V, E, w)$ un graphe non orienté connexe pondéré
Sortie : le MST T
Début
MPI_init()
MPI_Status status ;
MPI_Comm_size(MPI_COMM_WORLD, &totalnodes) ;
MPI_Comm_rank(MPI_COMM_WORLD, &mynode) ;
Partitionner V en V_i pour $i = 1, \ldots,$ totalnodes ;
// Revient à diviser les indice de 1 à n sur totalnodes processeurs
// Chaque processeur détermine son MSF en local
$T\_local = \emptyset$;
for each $v \in V_i$ **do**
  // Créer un ensemble disjoint pour chaque sommet
  Make_Set(v) ;
end for
Trier les arêtes de E_i par ordre croissant des poids ;
for each $e = (u, v) \in E_i$ pris dans cet ordre **do**
  // Find_Set(v) retourne l'ensemble disjoint auquel appartient v
  if Find_Set(u) $\neq$ (v) **then**
    $T\_local = T\_local \cup \{e\}$;
    // Unir les deux MSFs
    Union(u, v) ;
  end if
end for
Fusionner les MSFs deux à deux jusqu'à ce qu'il ne reste qu'un processus
IDÈE du merge deux à deux : un proc a envoie son MSF local au proc b
MPI_Send(T_local, count, MPI_Datatype, b, tag, MPI_Comm comm) ;
MPI_Recv(T_recv, count, MPI_Datatype, a, tag, MPI_Comm comm,MPI_Status) ;
Le proc b calcul son nouveau MSF local à partir de $T\_local \cup T\_recv$.
À la dernière itération, le dernier proc calcule le MST T_global à partir de son MSF T_local et du MSFs reçu T_recv
MPI_Finalize() ;
return T_global ;

Algorithm 3: Algorithme de Kruskal en parallèle

```
Entrée : G = (V, E, w) un graphe non orienté connexe pondéré
Sortie : le MST T
Début
T = ∅ ;
for each v ∈ V do
    // Créer un ensemble disjoint pour chaque sommet
    Make_Set(v) ;
end for
Trier les arêtes de E par ordre croissant des poids ;
for each e = (u, v) ∈ E pris dans cet ordre do
    // Find_Set(v) retourne l'ensemble disjoint auquel appartient v
    if Find_Set(u) ≠ (v) then
        T = T ∪ {e} ;
        // Unir les deux MSFs
        Union(u, v) ;
    end if
end for
return T ;
```

Algorithm 4: Algorithme séquentiel de Kruskal - Approche 2

Solution

IV-A-1. Montrons que la distance de Hamming satisfait l'inégalité triangulaire :
Soit $D_1(p,q)$ et $D_2(p,q)$ deux distances métriques, montrons que $D(p,q) = D_1(p,q)+D_2(p,q)$ est une distance métrique. Pour cela, on doit vérifier les trois axiomes :

Identité des indiscernables. $D(p,q) \geq 0$ avec égalité si et seulement si $p = q$.
Preuve : $D(p,q) = D_1(p,q)+D_2(p,q) \geq 0$ et égal à 0 ssi $p = q$ puisque D_1 et D_2 satisfont l'identité des indiscernables.

Symétrie. $D(p,q) = D(q,p)$.
Preuve : $D(p,q) = D_1(p,q) + D_2(p,q) = D_1(q,p) + D_2(q,p) = D(q,p)$ puisque D_1 et D_2 satisfont l'axiome de la symétrie.

Inégalité triangulaire. $D(p,q) \leq D(p,r) + D(r,q)$.
Preuve : $D(p,q) = D_1(p,q) + D_2(p,q) \leq D_1(p,r) + D_1(r,q) + D_2(p,r) + D_2(r,q) = D(p,r) + D(r,q)$ puisque D_1 et D_2 satisfont l'axiome de l'inégalité triangulaire.

Puisque la distance de Hamming est une distance séparable, c'est-à-dire une somme de d distances élémentaires de Hamming 1D, il suffit simplement de vérifier que la distance de Hamming 1D est une métrique satisfaisant l'inégalité triangulaire (puisque la somme de deux distances métriques est une distance métrique). Soient 3 bits $(p,q,r) \in \{0,1\}^2$, montrons que $D_H(p,q) + D_H(q,r) \geq D_H(p,r)\ \forall p,q,r$. Puisque $D_H(p,q) = D_H(q,p)$, on peut consider $p \leq q$ et on vérifie manuellement les six cas suivants :

p	q	r	$D_H(p,q) + D_H(q,r)$	$D_H(p,r)$
0	0	0	0	0
0	1	0	2	0
1	1	0	1	1
0	0	1	1	1
0	1	1	1	1
1	1	1	0	0

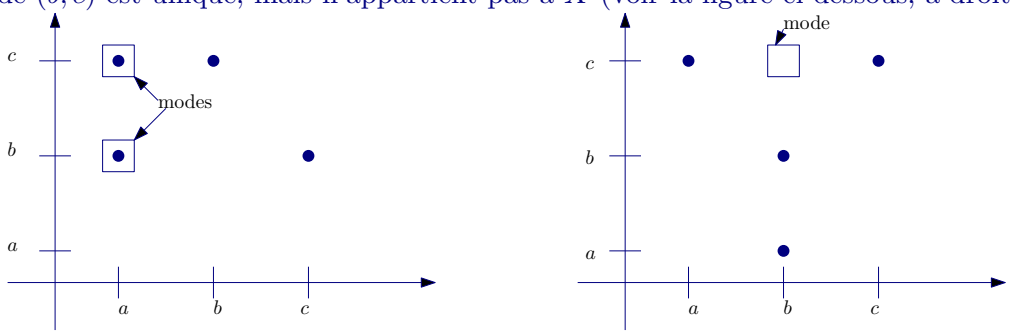
IV-A-2. Soit c un minimiseur de

$$l(x;X) = \frac{1}{n}\sum_{i=1}^{n} D_H(x,x_i).$$

On cherche à caractériser $c = \arg\min_{x\in E_1\times\ldots\times E_d} l(x;X)$.
On a :

$$\begin{aligned}\sum_{i=1}^{n} D_H(x,x_i) &= \sum_{i=1}^{n}\sum_{j=1}^{d} D_H(x^{(j)},x_i^{(j)}),\\ &= \sum_{j=1}^{d}\underbrace{\sum_{i=1}^{n} D_H(x^{(j)},x_i^{(j)})}_{=n_j(x)}.\end{aligned}$$

Puisque $n_j(x) \leq n_j^*$ avec égalité quand $x^{(j)} \in T_j$, en notant $f_j^* = \frac{n_j^*}{n}$, on en déduit qu'un mode de X doit avoir *pour chaque attribut* j une étiquette de *fréquence maximale* f_j^* de X. L'algorithme consiste donc à calculer pour chaque dimension j, les étiquettes de fréquence maximale T_j, et à choisir $c^{(j)}$ dans T_j, en temps linéaire. On peut aussi calculer combien il y a de modes maximaux : $\prod_{j=1}^d |T_j|$.
Le mode de $X = \{(a,b),(a,c),(c,b),(b,c)\}$ n'est pas forcément unique puisque nous avons deux minimiseurs (a,b) et (a,c) (voir la figure ci-dessous, à gauche). Pour $X = \{(a,c),(b,a),(b,b),(c,c)\}$, le mode (b,c) est unique, mais n'appartient pas à X (voir la figure ci-dessous, à droite).



IV-A-3. On peut ainsi généraliser l'algorithme de Lloyd des k-moyennes à la distance de Hamming. La fonction de coût à minimiser pour k clusters est :

$$l(X;C) = \frac{1}{n}\sum_{i=1}^{n} \min_{j\in[k]} D_H(c_j, x_i) \geq 0$$

où c_j sont les k centres de Hamming, les k modes. On initialise les k modes en tirant des étiquettes aléatoires pour chaque attribut. A chaque cycle, on associe les donnés à leur plus proche mode, puis pour chaque groupe de données ayant le même mode, on calcule le nouveau mode, et on termine ces itérations lorsque la fonction de coût ne décroit plus strictement.
Puisque $\forall x, x', \quad D_H(x,x') \leq \sum_{j=1}^d m_j = M$, on a initialement $nl(X;C) \leq nM$, et on assure à chaque itération de décroitre d'au moins une unité $nl(X;C)$ avec $l(X;C) \geq 0$. On a donc au plus nM itérations avant la convergence.
Voir [2] (1998) pour le papier introduisant les k-modes.

IV-B-1. Le centre de masse pour des poids positifs non-normalisés est :

$$c = \sum_{i=1}^{n} \frac{w_i}{W} x_i, \quad W = \sum_{i=1}^{n} w_i$$

Preuve : Soit on normalize les poids avec $w_i' = \frac{w_i}{W}$, et on en déduit ainsi immédiatement la formule, ou soit on minimise $\sum_{i=1}^n w_i \|x_i - c\|^2$, calcule le gradient et on l'annule en vérifiant que le Hessien est bien positif-défini (ce qui assure que la solution est unique).
Il en résulte que :

$$c_j = \frac{\sum_{i=1}^n w_{i,j}^p x_i}{\sum_{i=1}^n w_{i,j}^p}$$

IV-B-2. Pour les données X et les centres C fixés, notons $L(w,\lambda) = J_p(X;C;w) + \sum_{i=1}^n \lambda_i (1 - \sum_{j=1}^k w_{i,j})$ la fonction Lagrangienne qui prend en compte les n contraintes de poids à satisfaire : $\sum_{j=1}^k w_{i,j} = 1$.

Les dérivée partielles du Lagrangien pour les nk poids $w_{i,j}$ sont :

$$\frac{\partial L(w,\lambda)}{\partial w_{i,j}} = p w_{i,j}^{p-1} d_{i,j}^2 - \lambda_i, \quad \forall i \in [n] \forall j \in [k],$$

et elles s'annullent pour les valeurs des poids :

$$w_{i,j} = \left(\frac{\lambda_i}{p d_{i,j}^2} \right)^{\frac{1}{p-1}},$$

Puisque nous avons $\sum_{j=1}^k w_{i,j} = 1$, on en déduit que :

$$\sum_{j=1}^k \left(\frac{\lambda_i}{p d_{i,j}^2} \right)^{\frac{1}{p-1}} = 1.$$

On trouve donc :

$$\lambda_i = \left(\sum_{i=1}^k (p d_{i,j}^2)^{-\frac{1}{1-p}} \right)^{1-p},$$

et finalement on trouve :

$$w_{i,j} = \frac{d_{i,j}^{-\frac{2}{1-p}}}{\sum_{l=1}^k d_{i,l}^{-\frac{2}{1-p}}} = \frac{\|x_i - c_j\|^{-\frac{2}{1-p}}}{\sum_{l=1}^k \|x_i - c_l\|^{-\frac{2}{1-p}}}$$

IV-B-3. IV-B-3-a. A chaque itération, calculer les k centroïdes font nécessairement diminuer la fonction de coût : $J_p(X; C^{(t)}; w^{(t)}) \leq J_p(X; C^{(t+1)}; w^{(t)})$. puis la mise à jour des poids que l'on résout comme un problème de minimisation font également diminuer la fonction de coût : $J_p(X; C^{(t+1)}; w^{(t)}) \leq J_p(X; C^{(t+1)}; w^{(t+1)})$ et donc par transivitivé, on a : $J_p(X; C^{(t)}; w^{(t)}) \leq\leq J_p(X; C^{(t+1)}; w^{(t+1)})$. Puisque $J_p(X; C^{(t)}; w^{(t)}) \geq 0$, l'algorithme des k-moyennes floues converge monotonement vers un minimum local.

IV-B-3-b. Quand $p \to \infty$, puisque $\frac{1}{d_{i,j}^{-\frac{2}{p-1}}} \to 1$ (puisque $x^{\frac{1}{p}} \to 1$), on a $w_{i,j} \to \frac{1}{k}$.

IV-B-3-c. Quand $p \to 1^+$, $w_{i,j} = \frac{(d'_{i,j})^q}{\sum_{l=1}^k (d'_{i,l})^q}$ avec $d'_{i,j} = \frac{1}{d_{i,j}}$ et $q \to 0$. Donc $w_{i,j} \to \frac{(d'_{i,j})^q}{\max_{l \in [k]} (d'_{i,l})^q}$. Ainsi $w_{i,j^*} \to 1$ pour $j^* = \arg\max_{j \in [k]} d'_{i,j}$ et $w_{i,j'} \to 0$ pour $j' \neq 0$ puisque $\sum_{l=1} k w_{i,l} = 1$. Puisque $j^* = \arg\max_{j \in [k]} d'_{i,j} = \arg\min_{j \in [k]} d_{i,j}$, on en déduit que le point x_i est associé au centre le plus proche. Ainsi on a prouvé que les k-moyennes floues tendent vers les k-moyennes quand $p \to 1$ (avec $w_{i,j} \in \{0, 1\}$).

IV-B-4. La figure 2 donne le pseudo-code pour les k-moyennes floues en MPI.

La grande différence avec l'algorithme des k-moyennes parallèle en MPI est qu'on ne diffuse pas (en MPI, une opération de *broadcasting*, `Bcast`) à chaque itération les centroïdes à tous les processus.

Voir le papier [4] (2002) pour une implémentation détaillée en MPI.

Algorithm 1: Parallel Fuzzy c-Means (PFCM)

```
1:   P = MPI_Comm_size();
2:   myid = MPI_Comm_rank();
3:   randomise my_uOld[j][i] for each x[i] in fuzzy cluster j;
4:   do {
5:       myLargestErr = 0;
6:       for j = 1 to c
7:           myUsum[j] = 0;
8:           reset vectors my_v[j] to 0;
9:           reset my_u[j][i] to 0;
10:      endfor;
11:      for i = myid * (n / P) + 1 to (myid + 1) * (n / P)
12:          for j = 1 to c
13:              update myUsum[j];
14:              update vectors my_v[j];
15:          endfor;
16:      endfor;
17:      for j = 1 to c
18:          MPI_Allreduce(myUsum[j], Usum[j], MPI_SUM);
19:          MPI_Allreduce(my_v[j], v[j], MPI_SUM);
20:          update centroid vectors:
                     v[j] = v[j] / Usum[j];
21:      endfor;
22:      for i = myid * (n / P) + 1 to (myid + 1) * (n / P)
23:          for j = 1 to c
24:              update my_u[j][i];
25:              myLargestErr = max{|my_u[j][i] - my_uOld[j][i]|};
26:              my_uOld[j][i] = my_u[j][i];
27:          endfor;
28:      endfor;
29:      MPI_Allreduce(myLargestErr, Err, MPI_MAX);
30:  } while (Err >= epsilon)
```

FIGURE 2 – Les k-moyennes floues en MPI.

IV-C-1. Le maximum de vraisemblance est $\hat{\mu} = \arg\max_{\mu \in \mathbb{R}^d} \prod_{i=1}^n p(x_i; \mu)$. Puisque le logarithm est une fonction monotone croissante, la maximisation de la vraisemblance est équivalente à la maximisation de sa log-vraisemblance :

$$\hat{\mu} = \arg\max_{\mu \in \mathbb{R}^d} \sum_{i=1}^n l(x_i; \mu)$$

On a

$$l(x; \mu) = -\frac{1}{2\sigma^2}\|x - \mu\|^2 - d \log \sigma - -\frac{d}{2}\log(2\pi)$$

Puisque $\sigma > 0$ et d sont constants, $l(x; \mu) \propto \|x - \mu\|^2$, et ce problème de maximisation de vraisemblance est équivalent à :

$$\hat{\mu} = \arg\max_{\mu \in \mathbb{R}^d} -\sum_{i=1}^n \|x_i - \mu\|^2.$$

C'est donc équivalent à $\arg\min_{\mu \in \mathbb{R}^d} \sum_{i=1}^n \|x_i - \mu\|^2$. On reconnait la fonction à minimiser pour le centroïde, et on en déduit que :

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^n x_i.$$

IV-C-2. Quand $\sigma \to 0$, on a :

$$l_{j,i} = \frac{\exp(-\frac{1}{2\sigma^2}\|x_i - \mu_j\|^2)}{\sum_{l=1}^k \exp(-\frac{1}{2\sigma^2}\|x_i - \mu_l\|^2)}$$

Soit $j^* = \arg\max_l -\frac{1}{2\sigma^2}\|x_i - \mu_l\|^2$, alors quand $\sigma \to 0$, on a

$$l_{j,i} \to \frac{\exp(-\frac{1}{2\sigma^2}\|x_i - \mu_j\|^2)}{\exp(-\frac{1}{2\sigma^2}\|x_i - \mu_{j^*}\|^2)} \to \begin{cases} 1 & j = j^* \\ 0 & j \neq j^* \end{cases}$$

L'algorithme de Lloyd pour les k-moyennes est garanti de converger vers un minimum local en un temps fini. Puisqu'on ne répète jamais deux fois les mêmes partitions, le nombre d'itérations est borné par le deuxième nombre de Stirling $S(n,k) = \frac{1}{k!}\sum_{j=0}^k (-1)^{k-j}\binom{k}{j} j^n$. L'algorithme EM bien que convergeant monotonement ne finit pas en temps fini. Le nombre d'itérations est donc infini quand $\epsilon = 0$, et c'est pour cela qu'on a **besoin** d'implémenter un critère $\epsilon > 0$ pour stopper les itérations.

IV-C-3. Voici le pseudo-code de EM en MPI :

Plus de détails dans le papier [3].

Références

[1] Lih-Hsing Hsu, Rong-Hong Jan, Yu-Che Lee, Chun-Nan Hung, and Maw-Sheng Chern. Finding the most vital edge with respect to minimum spanning tree in weighted graphs. *Information Processing Letters*, 39(5) :277–281, 1991.

[2] Zhexue Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3) :283–304, 1998.

[3] Ayush Kapoor, Harsh Hemani, N. Sakthivel, and S. Chaturvedi. *Emerging ICT for Bridging the Future - Proceedings of the 49th Annual Convention of the Computer Society of India CSI Volume 2*, chapter MPI Implementation of Expectation Maximization Algorithm for Gaussian Mixture Models, pages 517–523. Springer International Publishing, Cham, 2015.

[4] Terence Kwok, Kate Smith, Sebastián Lozano, and David Taniar. Parallel fuzzy c-means clustering for large data sets. In *Euro-Par Parallel Processing*, pages 365–374. Springer, 2002.