# Algorithmes de flux

PAR C.DÜRR - UPMC - M1 4I405 CPA

10 mars 2016 (version 2)

## 1 Introduction

Les algorithmes de flux ont pris de l'importance avec la collecte massive de données, par exemple dans les serveurs webs ou dans les caisses enregistreuses des grandes surfaces. La difficulté réside dans le fait que les données sont tellement volumineuses, qu'il est impossible à un programme de les stocker entièrement en mémoire. Cette situation demande la conception d'algorithmes dont la complexité mémoire est sous-linéaire en la taille de l'entrée, idéalement logarithmique.

Dans la suite on considère un flux de données sous la forme $\sigma = x_1, ..., x_n \in [m]$, où $[m] := \{0, ..., m-1\}$ représente le domaine des éléments, qui sont supposés être des entiers pour plus de simplicité. L'algorithme doit être sous-linéaire en $n$ et en $m$, souvent polylogarithmique. L'acces aux données est séquentiel, en général $n$ et $m$ sont inconnus avant la fin de la lecture. Idéalement l'algorithme devrait faire une seule passe sur l'entrée.

Dans la suite nous présentons des algorithmes basiques pour des problèmes élémentaires. Une bonne introduction complète sur ces algorithme est le livre de Mathukrishnan [7].

### 1.1 Échantillonner

Le but est de choisir un élément $s$ du flux uniformément au hasard. La difficulté réside dans le fait qu'on ne connaisse pas le nombre d'éléments $n$ en avance.

| **Algorithme échantillonner uniformément** |
|---|
| Sur la lecture de $x_i$, choisir $s := x_i$ avec probabilité $1/i$. |

Soit $S$ la variable aléatoire associé à la variable $s$. L'invariant de l'algorithme est que $S$ est une distribution uniforme sur la partie du flux déjà vue. C'est vrai pour $i = 1$, trivialement, car $s = x_1$ avec probabilité 1. Puis en général, après avoir vu $n$ éléments, on a $\mathbb{P}[S = x_n] = 1/n$, par construction, et $\mathbb{P}[S = x_i] = 1/n$ pour tout $i \in [n-1]$, car la probabilité restante de $(n-1)/n$ est distribuée uniformément sur les $n-1$ éléments précedents par l'hypothèse de l'invariant.

Pour échantilloner $k$ éléments, il faut une mémoire de $O(k \log n)$ bits, ce qui est optimal, car c'est la taille que nécessite le stockage de la sortie de l'algorithme.

### 1.2 Échantillonner dans une fenêtre glissante

Imaginons à présent qu'on n'est plus intéressé dans les éléments trop anciens, et on veut choisir unformément au hasard un élément $s$ parmi les $w$ derniers éléments lu. De nouveau le problème est facile si on s'autorise de stocker $w$ éléments, mais on veut avoir une complexité logarithmique aussi en $w$.

| **Algorithme échantillonner dans une fenêtre glissante de taille $w$** |
|---|
| Choisir pour chaque élément de la fenêtre une valeur $v_i$ aléatoire uniformément choisie dans $[0, 1]$.<br>L'élément échantillonné parmi $x_{j-w+1}, ..., x_j$ est l'élément $x_i$ qui minimise $v_i$.<br>Il suffit de garder dans une liste $L$ seulement les éléments $x_i$ avec $v_i$ minimal parmi $v_i, ..., v_j$. |

Pour analyser la taille espérée de $L$, il faut déterminer la probabilité que le $k$-ème plus ancien élément – donc $x_{j-k+1}$ – soit dans $L$. Il s'agit de la probabilité que sa valeur soit minimale parmi les $k$ valeurs des $k$ éléments les plus anciens. Et cette probabilté est simplement $1/k$.

La taille espérée de $L$ est donc

$$\frac{1}{w} + \frac{1}{w-1} + \cdots + \frac{1}{2} + \frac{1}{1}$$

qui est le $w$-ème nombre harmonique et donc $O(\log w)$. La complexité espérée en mémoire de l'algorithme est donc de $O(\log w \log n)$ bits.

### 1.3 Exercices

1. On vous donne un flux de taille $n-1$ contenant les entiers de 0 à $n-1$, à l'exception d'un entier $x$. Le but est de trouver $x$ en une seule passe avec une mémoire de taille $O(\log n)$.

2. On vous donne un flux de taille $n$ d'entiers de $[m]$, avec la promesse qu'il existe une valeur $x$ qui apparaît au moins $\lceil n/2 \rceil$ fois. Le but est de trouver $x$ en une seule passe avec une mémoire de taille $O(\log n)$.

## 2 Vecteur de fréquences

Considérons un flux d'entiers $x_1, ..., x_n \in [m]$.

Pour chaque valeur $v \in [m]$ du domaine du flux on note $f_v$ le nombre d'apparaitions de $v$ dans le flux, donc $f_v := |\{i : x_i = v\}|$. Plusieurs stastiques peuvent être extraits de ce vecteur de fréquences.

**F0 ou le nombre d'éléments distincts.** Aussi noté par $\sum_{v \in [m]} f_v^0$.

**F1 ou la taille du flux.** Aussi noté par $\sum_{v \in [m]} f_v$. Ici on cherche des algorithmes dont la complexité en espace est $O(\log \log n)$.

**F2 ou l'élément de surprise.** Aussi noté par $\sum_{v \in [m]} f_v^2$. Permet de détecter des augmentations de fréquences importantes dans le flux, par rapport à la fréquence moyenne.

**F$\infty$ ou les éléments les plus fréquents.** Aussi noté par $\text{argmax}_{v \in [m]} f_v$.

# 3 Le nombre d'éléments distincts — F0

Imaginons la situation suivante. Vous disposez du fichier log d'un serveur web qui comporte les adresses IP (version 4) d'un milliard de connexions. Et vous aimeriez bien connaître le nombre de visiteurs distincts sur le site, donc le nombre d'adresses IP distincts dans le flux. Parfois des visiteurs différents peuvent apparaître avec une même adresse IP d'un serveur proxy, mais mettons cet aspect de côté. Une adresse IP est un nombre écrit sur 4 octets, il faudrait donc potentiellement 4 Gigaoctets pour stocker toutes les adresses. Ceci élimine une résolution de ce problème avec une simple table de hashage.

Nous cherchons alors une solution qui utiliserait une mémoire logarithmique en $n$ – la taille de l'entrée – et en $m$ – la taille du domaine des éléments, $m = 2^{32}$ dans notre cas. Le prix à payer est que nous ne pouvons espérer qu'une réponse approchée à notre problème.

Concrètement si $d$ est le nombre d'éléments distincts du le flux, alors on dit qu'un algorithme est une **$(\varepsilon, \delta)$-approximation** si le nombre retourné $\hat{d}$ satisfait

$$\mathbb{P}[d/(1+\varepsilon) \leqslant \hat{d} \leqslant d(1+\varepsilon)] \geqslant 1-\delta.$$

La probabilité est prise sur des choix aléatoires que l'algorithme aurait pris pendant le calcul, typiquement dans la phase d'initialisation. Par contre aucune hypothèse n'est prise sur le flux, et la condition mentionnée ci-haut doit être préservée par toute permutation sur les éléments du flux.

Ce relachement est nécessaire, car on peut montrer qu'un espace linéaire est nécessaire si on se restreint à des algorithmes déterministes ($\delta = 0$) ou des algorithmes exacts ($\varepsilon = 0$).

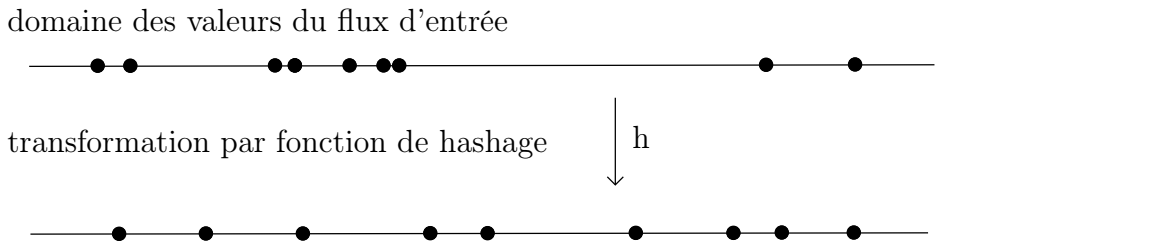


**Figure 1.** Principe de l'estimation des éléments distincts. Une fonction de hashage transforme un flux de valeurs arbitraires en un flux de valeurs aléatoires uniformes.

Plusieurs algorithmes ont été proposés pour ce problème. Une première solution a été trouvée par Flajolet et Martin en 1985 [4], mais nécessitait une famille de fonctions de hashage $\mathcal{H}$ avec $n$-indépendance des valeurs de $h$ choisie dans $\mathcal{H}$. Ceci est une condition trop forte sur $\mathcal{H}$ et aucune construction efficace d'une telle famille n'est connue pour l'instant.

En 1999 Alon, Matias et Szegedy[1] ont montré comment simplifier cette méthode et demander seulement la 2-indépendance. Puis notre présentation termine par un algorithme de 2004 de Bar-Yossef, Jayram, Kumar, Sivakumar et Trevisan [2]. La littérature sur ce problème est très riche, et pourrait remplir un cours entier. Donc la sélection des résultats pour ce cours n'est pas complète, et nous vous invitons à lire [5] pour avoir un état de l'art détaillé.

L'ingrédient principal à ces algorithmes est une bonne famille de fonctions de hashage.

## 3.1 Famille de fonctions de hashage 2-universelle

Soit $[M]^{[m]}$ l'ensemble des fonctions de $[m]$ vers $[M]$. Pour l'application que nous avons en tête $m = 2^{32}$, et $M \geq m$ sera précisé plus tard. Une famille $\mathcal{H} \subseteq [M]^{[m]}$ est 2-universelle, si la propriété suivante est satisfaite, où $h$ est choisie uniformément au hasard dans $\mathcal{H}$:

$$\forall x, x' \in [m] \, \forall y, y' \in [M] \, , x \neq x' \ : \ \mathbb{P}_h[h(x) = y \wedge h(x') = y'] = \frac{1}{M^2}.$$

En d'autres mots on veut que $h(x)$ soit uniforme sur $[M]$ pour tout $x$ et pour $x \neq x'$ on ait indépendance entre $h(x)$ et $h(x')$.

Une telle famille de fonctions a été proposée par **Carter et Wegman** en 1979, et elle envoie $[m]$ dans $[p]$ pour $p \geqslant m$ un nombre premier. Tous les entiers $0 \leqslant a, b \leqslant p-1$ définissent la fonction

$$h_{\mathrm{ab}}(x) = (ax + b) \bmod p.$$

La famille ainsi obtenue est 2-universelle, car le système $ax + b = y \wedge ax' + b = y'$ a une unique solution $(a, b)$ dans les entiers $\mathbb{Z}_p$ pour $x \neq x'$. Donc la probabilité de l'évènement $h(x) = y \wedge h(x') = y'$ est précisément $1/M^2$. On pourrait restreindre le choix de $a$ à $a \neq 0$, pour obtenir des fonctions de hashage sans collision, mais il faut alors travailler avec une variante de la notion de 2-universalité.

Certains algorithmes ont besoin que $M$ soit une puissance de 2. Pour obtenir une valeur dans $[M]$ il ne suffit pas de prendre ensuite le résultat dans $[p]$ modulo $M$, car sinon les $p - M$ premières valeurs ont une probabilité deux fois plus grande que le reste du domaine $[M]$.

Nous proposons alors de travailler avec une autre famille de fonctions de **hashage basée sur le ou-exclusif**, qui par contre a une complexité en espace plus grande. Soit $m = 2^k$, et Soit $\mathcal{H}$ l'ensemble de tous les matrices 01 de dimension k×k. Une matrice $H \in \mathcal{H}$ définit la fonction $h \colon x \mapsto Hx$, où $x$ est vu comme un vecteur 01 de dimension $k$. Concrètement $x$ est envoyé vers la somme des $h_i$ sur tous les $i \in [k]$ tel que $x_i = 1$ où $h_i$ est la i-ième colonne de $H$.

Par exemple considérons $k = 4$ et

$$H = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

La fonction de hashage défini par $H$ envoie $x = 6 = 0110_2$ sur $7 = 0111_2$ qui est la somme (modulo 2) des deux lignes du milieu de $H$, telles qu'elles sont sélectionées par $x$, donc $1010_2 \oplus 1101_2$.

Les fonctions $h$ de cette famille satisfont $h(x \oplus x') = h(x) \oplus h(x')$. Aussi si $x, x'$ sont tels qu'il n'existe pas de $i$ avec $x_i = 1$ et $x'_i = 1$ alors $h(x)$ et $h(x')$ sont des des valeurs aléatoires indépendants, car leur valeur dépend d'entrées disjointes de $H$. Avec ces observations on peut montrer que cette famille est 2-universelle: Soit $z = x \wedge x'$ tel que $z_i = 1$ si et seulement si $x_i = 1$ et $x'_i = 1$. Alors pour $x, x', y, y' \in [m]$ et $x \neq x'$ on a

$$\begin{aligned} \mathbb{P}[h(x) = y \wedge h(x') = y'] &= \mathbb{P}[h(x \oplus z) = y \oplus h(z) \wedge h(x' \oplus z) = y' \oplus h(z)] \\ &= \frac{1}{2^{2k}}. \end{aligned}$$

Une remarque pratique. La fonction interne hash de Python permet de transformer en entier tout objet donnée, typiquement une chaîne de caractères. C'est très pratique, mais sur les entiers cette fonction est juste l'identité. Il en est de même pour la méthode Integer.hashCode en Java.

## 3.2 L'algorithme de Alon, Matias et Szegedy

L'idée principale consiste à choisir aléatoirement une fonction de hashage $h \in \mathcal{H}$, et à l'appliquer à chaque élément du flux. S'il n'y a pas de collision avec $h$, alors on voit arriver exactement $d$ éléments choisis uniformément dans $[m]$. La valeur du plus grand de ces éléments doit alors donner une indication sur $d$.

Supposons que $m$ soit de la forme $2^k$. Pour $v \in [m]$ soit zeros($v$) le nombre de 0 consécutif finaux de l'expansion binaire de $v$. Par exemple zeros(40) = zeros($101000_2$) = 3, et zeros(0) = $k$.

**Algorithme AMS pour estimer le nombre éléments distincts**

L'algorithme suppose que $m$ soit de la forme $2^k$.
*Initialisation*
    Choisir au hasard une fonction de hashage $h$: $[m] \to [m]$ d'une famille 2-universelle.
    z:=0
*Traitement de j*
    $z := \max(z, \text{zeros}(h(j)))$
*Estimation*
    Retourner $\hat{d} := 2^{z+1/2}$.

L'algorithme utilise seulement $O(\log m)$ bits. L'intuition de cet algorithme est la suivante. Si les valeurs de hashage sont bien distribuées sur $[m]$ alors on s'attend à ce que la plus petite aie la valeur $m/(d+1)$. Mais pour l'analyse, plutôt que de travailler avec les véritables valeurs de hashage, on considère des arrondis grossiers, basés sur le nombre de zéros consécutifs dans l'expansion binaire.

Pour implémenter efficacement cet algorithme, on pourrait compter à la place du nombre de zéros consécutif *finaux* le nombre de 0 consécutifs *initiaux*. Ce changement est valide, car il correspond à appliquer une bijection sur $h$, ce qui préserve les propriétés de la famille de fonctions de hashage. Il suffit alors de déterminer la valeur minimale $y$ sur toutes les valeurs de hashage, et poser $z = k - 1 - \lfloor \log_2(y) \rfloor$ si $y > 0$.

Pour analyser formellement la qualité de l'estimateur, on introduit une variable aléatoire indicatrice $X_{r,j}$ pour chaque $j \in [m]$ et entier $r \geqslant 0$, pour l'évènement $\text{zeros}(h(j)) \geqslant r$, où la probabilité est prise sur le choix de la fonction de hashage $h$. Soit $Y_r := \sum_{j:f_j>0} X_{r,j}$, où $f_j$ est le nombre d'occurences de $j$ dans le flux. Soit $t$ la valeur de $z$ à la fin de l'algorithme. Alors

$$Y_r > 0 \;\Leftrightarrow\; t \geqslant r$$

et de manière équivalente

$$Y_r = 0 \;\Leftrightarrow\; t \leqslant r - 1.$$

Comme chacun des $\log_2 m$ bits de $h(j)$ est choisi unformément sur $\{0, 1\}$ on a

$$\mathbb{E}[X_{r,j}] \;=\; \mathbb{P}[\text{zeros}(h(j)) \geqslant r] = 1/2^r.$$

Nous pouvons désormais estimer l'espérance et la variance de $Y_r$. C'est ici que la 2-universalité de de la famille de fonctions de hashage est utilisée.

$$\begin{aligned}
\mathbb{E}[Y_r] &= \sum_{j:f_j>0} \mathbb{E}[X_{r,j}] = \frac{d}{2^r} && \text{(par linéarité de } l'\text{espérance)} \\
\mathbb{V}\text{ar}[Y_r] &= \sum_{j:f_j>0} \mathbb{V}\text{ar}[X_{r,j}] && \text{(par la 2 à 2 indépendance)} \\
&\leqslant \sum_{j:f_j>0} \mathbb{E}[X_{r,j}^2] && \text{(par } d\text{éfinition de la variance)} \\
&= \sum_{j:f_j>0} \mathbb{E}[X_{r,j}] && \text{(par } 1^2 = 1 \text{ et } 0^2 = 0) \\
&= \frac{d}{2^r}.
\end{aligned}$$

Désormais nous pouvons borner la probabilité que l'algorithme produit une estimation au moins $2^{r+1/2}$ ou au plus cette valeur, à l'aide respectivement des inégalités de Markov et de Chebychev.

$$\begin{aligned}
\mathbb{P}[Y_r > 0] &= \mathbb{P}[Y_r \geqslant 1] \leqslant \frac{\mathbb{E}[Y_r]}{1} = \frac{d}{2^r} \\
\mathbb{P}[Y_r = 0] &= \mathbb{P}[Y_r - \mathbb{E}[Y_r] = -\mathbb{E}[Y_r]] \\
&= \mathbb{P}[Y_r - \mathbb{E}[Y_r] = -d/2^r] \\
&= \mathbb{P}[|Y_r - \mathbb{E}[Y_r]| = d/2^r] \\
&\leqslant \mathbb{P}[|Y_r - \mathbb{E}[Y_r]| \geqslant d/2^r] \leqslant \frac{\mathbb{V}\text{ar}[Y_r]}{(d/2^r)^2} \leqslant \frac{2^r}{d}.
\end{aligned}$$

Pour borner la probabilité que l'algorithme surestime $d$ avec un facteur 3, soit $a$ le plus petit entier tel que $2^{a+1/2} \geqslant 3d$. Nous avons

$$\mathbb{P}[\hat{d} \geqslant 3d] \;=\; \mathbb{P}[Y_a > 0] \leqslant \frac{d}{2^a} \leqslant \frac{\sqrt{2}}{3}$$

et pour $b$ le plus grand entier tel que $2^{b+1/2} \leqslant d/3$ nous avons

$$\mathbb{P}[\hat{d} \leqslant d/3] \;=\; \mathbb{P}[Y_{b+1} = 0] \leqslant \frac{2^{b+1}}{d} \leqslant \frac{\sqrt{2}}{3}.$$

La probabilité que l'estimation $\hat{d}$ ne satisfaisse pas $d/3 \leqslant \hat{d} \leqslant 3d$ peut donc être majorée par $2\sqrt{2}/3 \leqslant 0.95$ et l'algorithme AMS est une $\left(2, 2\sqrt{2}/3\right)$-approximation.

## 3.3 Technique du médian

Pour diminuer la probabilité d'erreur de l'algorithme précédent à une probabilté $\delta$ donnée, il suffit d'éxécuter $O(\log(1/\delta))$ copies de l'algorithme en parallèle et de retourner le médian des estimations produites. Supposons qu'on exécute $2D+1$ copies. La probabilité que le médian soit une sur-estimation est au plus $\left(\sqrt{2}/3\right)^D$. Et cette probabilité est au plus $\delta/2$ si

$$\begin{aligned}
\left(\frac{\sqrt{2}}{3}\right)^D &\leqslant \delta/2 \\
\left(\frac{3}{\sqrt{2}}\right)^D &\geqslant 2/\delta \\
D &\geqslant \log_{3/\sqrt{2}}(2/\delta).
\end{aligned}$$

Pour le même choix de $D$, la probabilité d'une sous-estimation est également au plus $\delta/2$.

Cette technique permet d'obtenir une $(2,\ \delta)$-approximation avec une complexité espace de $O(\log(1/\delta)\log(m))$. La section suivante explique comment obtenir une $(\varepsilon, \delta)$-estimation pour $\varepsilon$ arbitrairement petit.

## 3.4 L'algorithme de Bar-Yossef, Jayram, Kumar, Sivakumar et Trevisan

L'algorithme suivant est un raffinement de l'algorithme AMS dans le sens où en plus d'observer la plus grande classe contenant une valeur de hashage, l'algorithme maintient l'ensemble des valeurs dans ces classes. Pour éviter de stocker trop de valeurs, les classes les moins importantes pour l'estimation sont vidées si nécessaire.

**Algorithme BJKST pour estimer le nombre d'éléments distincts**

Soit une constante $c := 12 \cdot 48$ expliquée plus tard
*Initialisation*
    Choisir au hasard une fonction de hashage $h$: $[m] \to [m]$ d'une famille 2-universelle.
    t:=0
    B:=un tableau d'ensembles vides
*Traitement de $j$*
    $z_j := \text{zeros}(h(j))$
    si $z_j \geqslant t$ alors
        $B[z_j] := B[z_j] \cup \{h(j)\}$
        tant que $|B| > c/\varepsilon^2$ faire
            B[t]:=$\emptyset$
            t:=t+1
*Estimation*
    Retourner $\hat{d} := |B| 2^t$.

Dans cet algorithme $|B|$ denote la taille totale de $B$, donc $\sum_{r=0}^{\log m} |B[r]|$.

Pour chaque valeur aléatoire $v$, la probabilité que $\text{zeros}(v) \geqslant t$ est $1/2^t$. Donc on s'attend à ce que parmi les $d$ valeurs distinctes, $d/2^t$ aient cette propriété. Ceci motive l'estimation donnée par l'algorithme.

Le même article de recherche propose un 3ème algorithme, qui au lieu de stocker les valeurs de hashage par $h$, stocke les valeurs de hashage par une 2ème fonction $g$ à image plus petite, afin d'économiser d'avantage de mémoire utilisée. Mais nous ne le présentons pas.

Pour la complexité en espace, on peut supposer que $1/\varepsilon^2$ soit seulement $o(m)$. Le nombre d'éléments dans $B$ est borné à $O(1/\varepsilon^2)$, et chaque élément nécessite $O(\log m)$ bits. Le stockage de la fonction de hashage nécessite également $O(\log m)$ bits normalement. Au total la complexité en espace est $O((\log m)/\varepsilon^2)$.

L'analyse de la qualité de l'estimation suit les mêmes étapes que pour l'algorithme AMS. Comme pour l'algorithme précédent nous avons

$$\begin{aligned} \mathbb{E}[Y_r] &= \frac{d}{2^r} \\ \mathbb{V}\text{ar}[Y_r] &= \frac{d}{2^r}. \end{aligned} \tag{1}$$

Observons que pour $t = 0$, l'algorithme n'a enlevé aucun élément de $B$ et l'estimation est exacte dans ce cas.

Pour le cas échéant, nous cherchons à borner la probabilité que l'estimation soit trop loin de la véritable valeur, donc la probabilité que

$$|Y_t 2^t - d| \geqslant \varepsilon d$$

ou de manière équivalente

$$\left|Y_t - \frac{d}{2^t}\right| \geqslant \frac{\varepsilon d}{2^t}.$$

Pour borner la probabilité de cet évènement, on va sommer sur $r$, mais utiliser une borne différente pour un petit $r$ que pour un grand $r$. Le seuil sera défini par l'unique entier $s$ tel que

$$\frac{12}{\varepsilon^2} \leqslant \frac{d}{2^s} \leqslant \frac{24}{\varepsilon^2}.$$

La probablité d'échec peut donc être formulée comme la somme suivante, qui débute à 1, car pour t=0 l'algorithme était sans erreur.

$$\begin{aligned}
& \sum_{r=1}^{\log m} \mathbb{P}\left[\left|Y_r - \frac{d}{2^r}\right| \geqslant \frac{\varepsilon d}{2^r} \wedge t = r\right] \\
\leqslant & \sum_{r=1}^{s-1} \mathbb{P}\left[\left|Y_r - \frac{d}{2^r}\right| \geqslant \frac{\varepsilon d}{2^r} \wedge t = r\right] + \sum_{r=s}^{\log m} \mathbb{P}[t = r] \\
= & \sum_{r=1}^{s-1} \mathbb{P}\left[|Y_r - \mathbb{E}[Y_r]| \geqslant \frac{\varepsilon d}{2^r} \wedge t = r\right] + \mathbb{P}[t \geqslant s] \\
= & \sum_{r=1}^{s-1} \mathbb{P}\left[|Y_r - \mathbb{E}[Y_r]| \geqslant \frac{\varepsilon d}{2^r} \wedge t = r\right] + \mathbb{P}[Y_{s-1} \geqslant c/\varepsilon^2]
\end{aligned}$$

où on a utilisé l'égalité (1). Désormais nous allons appliquer sur la première partie l'inégalité de Chebychev et sur la deuxième l'inégalité de Markov. Et donc nous pouvons majorer cette somme par

$$\begin{aligned}
& \sum_{r=1}^{s-1} \frac{\mathbb{V}\mathrm{ar}[Y_r]}{(\varepsilon d/2^r)^2} + \frac{\mathbb{E}[Y_{s-1}]}{c/\varepsilon^2} \\
\leqslant & \sum_{r=1}^{s-1} \frac{2^r}{\varepsilon^2 d} + \frac{\varepsilon^2 d}{c 2^{s-1}} \\
\leqslant & \frac{2^s}{\varepsilon^2 d} + \frac{2\varepsilon^2 d}{c 2^s} \\
\leqslant & \frac{\varepsilon^2}{12\varepsilon^2} + \frac{2\varepsilon^2 24}{c\varepsilon^2} \\
\leqslant & \frac{1}{6}.
\end{aligned}$$

La dernière inégalité est une conséquence du choix de $c$.

L'algorithme est donc une $(\varepsilon, 1/6)$-estimation, dont la probabilité d'erreur peut être diminuée avec la technique du médian, expliquée en section 3.3.

# 4 Trouver les éléments les plus fréquents

Le flux $x_1, ..., x_n \in [m]$ définit un vecteur de fréquences $f_1, ..., f_m$ tel que $f_j := |\{i: x_i = j\}|$. Un premier but est d'estimer le vecteur des fréquences. Le but ultime est de produire la liste des éléments dont la fréquence dépasse un certain seuil $n/k$ donné.

## 4.1 L'algorithme de Misra-Gries

L'algorithme de Misra-Gries [6] résoud ce problème, et peut être étendu avec une deuxième passe pour produire la liste des éléments dont la fréquence dépasse une fraction donnée du flux.

**Algorithme Misra-Gries(k)**

*Initialisation*
    A := dictionnaire vide
*Sur la lecture d'une valeur j*
    si j est une clé dans A
        incrémenter $A[j]$
    sinon si le nombre de clés dans A $< k-1$
        A[j]:=1
    sinon
        pour tous les clés $\ell$ dans A
            décrémenter $A[\ell]$
            si $A[\ell] = 0$, alors enlever $\ell$ de $A$
*Sur la requête d'une valeur a*
    Si $a$ est une clé dans A
        répondre $\tilde{f}_a = A[a]$
    sinon
        répondre $\tilde{f}_a = 0$

Pour la complexité en espace de l'algorithme, supposons que $A$ est implémenté sous forme d'un arbre binaire de recherche équilibré. Il y a au plus $k-1$ entrées dans $A$, une clé nécessite $\lceil \log m \rceil$ bits et une valeur $\lceil \log n \rceil$ bits au plus. La complexité en espace est donc $O(k(\log n + \log m))$.

Considérons maintenant la qualité de la réponse de l'algorithme. On note que chaque incrémentation de $A[j]$ est due à la présence d'une valeur $j$ dans le flux. Donc

$$\tilde{f}_a \leqslant f_a.$$

D'un autre côté, chaque décrémentation de $A[j]$ nécessite la présence de $k$ valeurs distincts dans $A$ et donc dans le fux. Leur trace dans $A$ est supprimée par la décrémentation, y compris celle de $j$. Le flux étant de longueur $n$, il ne peut y avoir plus que $n/k$ décrémentations. En conséquence

$$f_a - \frac{n}{k} \leqslant \tilde{f}_a \leqslant f_a.$$

Dans le problème FRÉQUENCE on chercher à obtenir la listes des éléments $a$ dont $f_a$ dépasse un seuil $n/k$ donné. Tous ces éléments se trouvent dans le dictionnaire $A$ à la fin de l'algorithme de Misra-Gries. Mais dans $A$ il peut y avoir aussi des éléments avec des fréquences plus petites. Une deuxième passe sur l'entrée permet alors de déterminer les fréquences exactes de clés dans $A$ et déterminer ceux dont la fréquence dépasse vraiment le seuil.

# 5 Clustering

Nous considérons un flux de points d'un espace M muni d'une métrique $d$. Par exemple un flux de points du plan $\mathbb{R}^2$ muni de la distance euclidienne. Une métrique est une fonction de distance $d$ qui satisfait les propriétés d'identité $d(x,x)=0$, de symmétrie $d(x,y)=d(y,x)$ et d'inégalité triangulaire $d(x,y)\leqslant d(x,z)+d(z,y)$ pour tout $x,y,z\in M$.

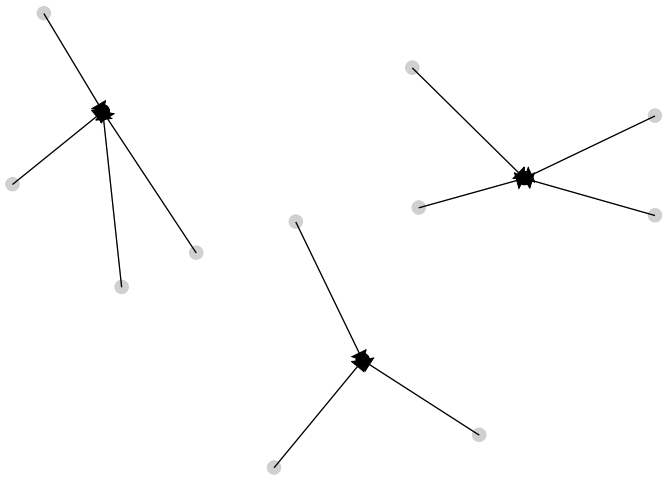

**Figure 2.** Une décomposition de points en 3 clusters. Les représentants sont illustrés en noir.

On doit partitionner ces points en $k$ ensembles appelés *clusters*. Ces clusters sont censés contenir des points proches les uns des autres. Concrètement chaque cluster est décrit par un *représentant* du cluster, qui est un des points du flux. Chaque point est associé au représentant le plus proche, avec un choix arbitraire en cas d'égalité (à la manière d'un *diagramme de Voronoi*, pour ceux qui connaissent). Cette affectation définit les $k$ clusters. Une bonne application est le placement de $k$ points de service dans un territoire. Chaque habitant va se déplacer au point de service le plus proche, et il est souhaitable que ces distances de déplacements soient petits.

Plusieurs objectifs ont été définis dans la littérature. Pour les introduire formellement nous étendons la fonction de distance $d$ aux ensembles en définissant pour tout $x\in M$ et $S\subseteq M$

$$d(x,S) := \min_{y\in S} d(x,y).$$

Soit $\sigma$ une séquence de points de $M$, et $R$ un choix d'au plus $k$ points de $\sigma$. Trois principales valeurs objectifs ont été considérées dans la littérature, à savoir

$$\begin{aligned} \Delta_\infty(\sigma, R) &:= \max_{x \in \sigma} d(x, R) && (k-\text{center}) \\ \Delta_1(\sigma, R) &:= \sum_{x \in \sigma} d(x, R) && (k-\text{median}) \\ \Delta_2(\sigma, R) &:= \sum_{x \in \sigma} d(x, R)^2. && (k-\text{means}) \end{aligned}$$

L'objectif $k$-center pourrait être motivé pour l'implantation d'hôpitaux sur un territoire, où on veut minimiser le pire temps de déplacement vers l'hôpital le plus proche. L'objectif $k$-median considère par contre le temps de déplacement moyen. Alors que l'objectif $k-$means a été étudié depuis les années 1960 dans le domaine des statistiques et de l'apprentissage par ordinateur. Dans un souci de simplicité dans ce cours nous allons considérer l'objectif k-center.

## 5.1 L'algorithme doublant

Il s'agit d'un algorithme à une seule passe qui maintient un seuil $\tau$. Celui-ci approxime la valeur objective $\Delta_\infty(\sigma,\ R)$ pour la partie de la séquence déjà vu, dans un sens que nous allons préciser plus tard.

**Algorithme doublant [3]**

*Phase d'initialisation* :
$S :=$ les premiers $k+1$ points du flux
Soient $x, y \in S$ deux points qui minimisent $d(x, y)$
$\tau := d(x, y)$
$R := S \setminus \{x\}$
*Phase de croisière : sur la lecture d'un élément $x$ du flux*
si $d(x, R) > 2\tau$
$R := R \cup \{x\}$
tant que $|R| > k$
$\tau := 2\tau$
R:=sous ensemble maximal $R' \subseteq R$ avec $d(y, z) \geqslant \tau$ pour $y, z \in R'$ et $y \neq z$.

Pour comprendre le comportement de l'algorithme, nous commençons par montrer un invariant qui est préservé tout au long du déroulement de l'algorithme.

**Proposition 1.** *L'algorithme maintient l'invariant suivant.*

1. *Il existe un ensemble $S$ avec $|S| = k+1$ et $d(x, y) \geqslant \tau/2$ pour tout $x, y \in S$.*

2. $\Delta_\infty(\sigma, R) \leqslant 2\tau$.

**Démonstration.** L'invariant 1 est satisfait à la fin de phase d'initialisation par construction, même pour une borne plus forte $d(x,\, y) \geqslant \tau$. Au début de chaque itération de la boucle tant que $R$ contient $k+1$ éléments et satisfait $d(x, y) \geqslant \tau$, pour tout $x, y \in R, x \neq y$. Donc quand $\tau$ est doublé, l'invariant 1 est préservé.

L'invariant 2 est satisfait après la phase d'initialisation, car à ce moment là $\Delta_\infty(\sigma,$ $R) = \tau$. Maintenant montrons que l'invariant 2 est préservé pendant la phase de croisière. Dans le cas où $d(x, R) \leqslant 2\tau$, ni $R$ ni $\tau$ sont modifiés. Donc l'invariant est préservé, car l'ajout de $x$ à $\sigma$ ne change pas $\Delta_\infty(\sigma, R) \leqslant 2\tau$.

Considérons le cas restant $d(x, R) > 2\tau$. Juste après l'ajout de $x$ à $R$, l'invariant 2 est préservé, car par $d(x, R) = 0$, l'objectif $\Delta_\infty(\sigma, R)$ n'augmente pas.

Qu'en est-il après la boucle tant que? Pour montrer que l'invariant 2 est préservé, ll nous reste à montrer que pour le choix $R'$ suivant, on ait $d(x, R') \leqslant 2\tau$ pour tout $x \in \sigma$.

Soit $y$ le représentant de $R$ qui était le plus proche de $x$. Si $y$ a survécu dans $R'$ — donc si $y \in R'$ — alors $d(x,\, R') = d(x,\, R) \leqslant \tau$ par l'invariant renforcé par le doublement de $\tau$.

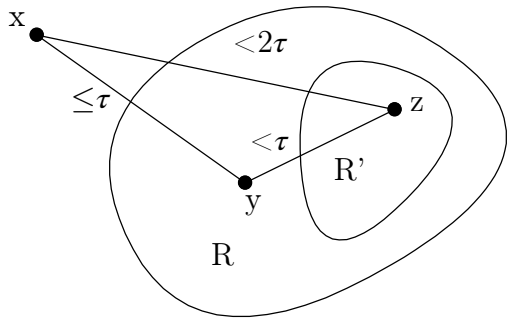


**Figure 3.** Idée de la preuve de préservation de l'invarant

Pour le cas restant, observons qu'il existe un point $z \in R'$ avec $d(y,\, z) < \tau$, sinon $R' \cup \{y\}$ aurait contredit la maximalité de $R'$. Ceci mène vers cette série d'inégalités

$$\begin{aligned} d(x, R') &\leqslant d(x, y) + d(y, z) \\ &< d(x, R) + \tau \\ &\leqslant 2\tau, \end{aligned}$$

et conclu la preuve. □

L'analyse de la performance de l'algorithme repose sur l'observation suivante.

**Proposition 2.** *Soit $x_1, \ldots, x_{k+1} \in \sigma \subseteq M$ tel que $d(x_i, x_j) \geqslant \tau$ pour tout $1 \leqslant i < j \leqslant k+1$. Alors pour tout $R \subseteq \sigma$ avec $|R| \leqslant k$ on a $\Delta_\infty(\sigma, R) \geqslant \tau/2$.*

Notez que l'énoncé est très fort, car à priori $R$ peut être complètement disjoint de $\{x_1, \ldots, x_{k+1}\}$.

**Démonstration.** La preuve repose sur le fait qu'il doit exister $1 \leqslant i < j \leqslant k+1$, tel que $x_i$ et $x_j$ sont associés au même représentant de $R$, disons $y$. Alors par l'inégalité triangulaire

$$\tau \leqslant d(x_i, x_j) \leqslant d(x_i, y) + d(x_j, y).$$

Donc une des distances $d(x_i, y), d(x_j, y)$ doit être au moins $\tau/2$. □

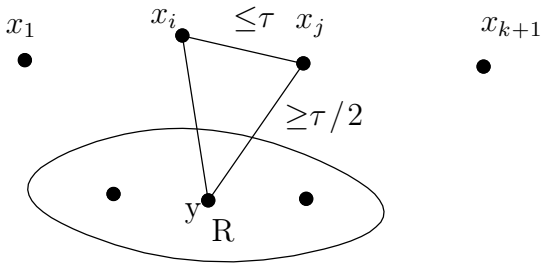


**Figure 4.** Idée de la preuve de la proposition 2

Maintenant nous avons tous les ingrédients pour montrer le théorème suivant.

**Théorème 3.** *L'algorithme doublant produit une solution dont la valeur objective est au plus 8 fois l'optimum.*

**Démonstration.** Soit $R^\star$ une solution optimale, et $R, \tau$ les valeurs produits par l'algorithme. Par l'invariant 2 nous avons $\Delta_\infty(\sigma, R) \leqslant 2\tau$. Mais par l'invariant 1 tous les points de $R$ sont à distance au moins $\tau/2$, et donc par la proposition 2, on a $\Delta_\infty(\sigma, R^\star) \geqslant \tau/4$. Les deux bornes prouvent que $\Delta_\infty(\sigma, R) \leqslant 8\Delta_\infty(\sigma, R^\star)$. □

# 6 Rappel de théorie de probabilité

Nous avons besoin d'analyser des algorithmes probabilistes et allons utiliser pour cela trois bornes importantes, l'inégalité de Markov,

## Espérance et variance

Si une donnée $X$ dépend d'un tirage aléatoire, on dit que $X$ est variable aléatoire et on la note par une lettre majuscule. Si $X$ peut prendre les valeurs dans un domaine $[m]$ on peut associer à chaque valeur $j \in [m]$ la probabilité $\mathbb{P}[X = j]$ que $X$ vaut $j$. On appèle $X = j$ un *évènement*. Donc $X$ peut être vue comme une distribution sur $[m]$. Deux mesures importantes d'une variable aléatoire sont l'*espérance* et la *variance*.

L'espèrance est la valeur moyenne qu'on attend de $X$, elle définie comme

$$\mathbb{E}[X] := \sum_{j \in [m]} j \cdot \mathbb{P}[X = j].$$

Une autre forme utile est $\mathbb{E}[X] = \sum_{j \in [m]} \mathbb{P}[X \geqslant j]$.

La variance mesure comment $X$ est concentré autour de l'espérance $\mathbb{E}[X]$. Elle est définie comme

$$\mathbb{V}\mathrm{ar}[X] := \mathbb{E}[(X - \mathbb{E}[X])^2].$$

Par exemple si $X$ est déterministe (associe probabilité 1 à une seule valeur $j \in [m]$) alors la variance de $X$ est 0. Une forme alternative (facile à dériver) est

$$\mathbb{V}\mathrm{ar}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2.$$

## Indépendance

On dit que deux variables aléatoires $X, Y$ sur le domaine $[m]$ sont indépendantes si et seulement si pour toutes les valeurs $i, j \in [m]$ on ait

$$\mathbb{P}[X = i, Y = j] = \mathbb{P}[X = i] \cdot \mathbb{P}[Y = j].$$

L'indépendance entre $X$ et $Y$ veut dire que la probabilité que $X$ soit égale à une valeur fixé est indépendant de l'évènement que $Y$ soit égale à une valeur fixée, formelement $X$ et $Y$ sont indépendant si pour tout $i, j \in [m]$ on ait

$$\mathbb{P}[X = i | Y = j] = \mathbb{P}[X = i].$$

La notation $X = i | Y = j$ décrit l'évènement $X = i$ dans le monde où $Y = j$, donc $\mathbb{P}[X = i | Y = j]$ est la probabilité que $X$ vaut $i$ sachant que $Y$ vaut $j$.

Plus généralement un groupe de variables $X_1$, ..., $X_n$ est $k$-indépendant pour un entier $k \in \{2, ..., n\}$ si et seulement si pour tout $i_1, ..., i_k \in [n]$ et $a_1, ..., a_k \in [m]$ on ait

$$\mathbb{P}[X_{i_1} = a_1, ..., X_{i_k} = a_k] = \mathbb{P}[X_{i_1} = a_1] \cdots \mathbb{P}[X_{i_k} = a_k].$$

La borne par union suivante s'applique même en cas de dépendance entre les variables, mais elle peut être très grossière.

$$\mathbb{P}[X_{i_1} = a_1, ..., X_{i_k} = a_k] \leqslant \mathbb{P}[X_{i_1} = a_1] + \cdots + \mathbb{P}[X_{i_k} = a_k].$$

### Linéarité

L'espérance est linéaire, à savoir $\mathbb{E}[\mathrm{cX}] = c\mathbb{E}[X]$ et $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$, pour deux variables aléatoires $X, Y$ qui peuvent même être dépendants.

Il existe aussi une propriété de linéarité de la variance pour deux variables aléatoires $X, Y$, mais ils doivent être indépendants,

$$\mathbb{V}\mathrm{ar}[X+Y] = \mathbb{V}\mathrm{ar}[X] + \mathbb{V}\mathrm{ar}[Y].$$

### Inégalité de Markov

Pour tout $c > 0$ on a $\mathbb{P}[X \geqslant c] \leqslant \mathbb{E}[X]/c$. On ne peut pas utiliser cette borne pour $\mathbb{P}[X \leqslant c]$.

### Inégalité de Chebychev

Pour tout $c > 0$ on a

$$\mathbb{P}[|X - \mathbb{E}[X]| \geqslant c] \leqslant \frac{\mathbb{V}\mathrm{ar}[X]}{c^2}.$$

### Borne de Chernoff

Soient $X_1$, ..., $X_k$ des variables aléatoires sur indépendants, décrivant une même distribution sur $[m]$, i.e. $\mathbb{P}[X_i = a] = \mathbb{P}[X_j = a]$ pour tout $1 \leqslant i < j \leqslant k$ et $a \in [m]$. Soit $\mu = \mathbb{E}[X_i]$ (peu importe $i$). Soit $X = \frac{1}{k}\sum_{i=1}^{k} X_i$, et $0 < \delta < 1$, alors

$$\mathbb{P}[|X - \mathbb{E}[X]| \geqslant \delta\mathbb{E}[X]] \leqslant 2\exp\left(-\frac{k\delta^2\mathbb{E}[X]}{3}\right).$$

On peut se servir de cette borne pour répéter plusieurs fois l'exécution d'un algorithme aléatoire et obtenir une meilleur estimation de la valeur retournée espérée.

# Bibliographie


**[1]** Noga Alon, Yossi Matias et Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58:137–147, 1999.

**[2]** Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar et Luca Trevisan. Counting distinct elements in a data stream. Dans *Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

**[3]** Moses Charikar, Chandra Chekuri, Tomas Feder et Rajeev Motwani. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing*, 33(6):1417–1440, 2004.

**[4]** Philippe Flajolet et G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

**[5]** Stefan Heule, Marc Nunkesser et Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. Dans *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

**[6]** Jayadev Misra et David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

**[7]** Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.