PROJET 8

APPROXIMATECOUNTING : Algorithmes probabilistes pour des flots de données

sujet proposé par Lucas Gerin

lucas.gerin@polytechnique.edu

Mots-clés : Algorithmes probabilistes, Probabilités conditionnelles, Calculs de lois.

L'algorithmique de flots de données (*stream algorithms*) est le domaine de l'informatique théorique qui s'intéresse à la résolution de problèmes algorithmiques simples mais avec la contrainte que des paquets de données arrivent très vite, sans qu'on puisse forcément toutes les stocker et les manipuler. On doit également utiliser un très petit espace de stockage en mémoire. Vues ces contraintes très fortes on relâche un peu l'objectif et on s'autorise à ne renvoyer qu'une valeur approchée de la solution plutôt que la solution exacte. Le développement des algorithmes de flots de données remonte aux années 1970, à l'époque où l'espace mémoire était une ressource très coûteuse. Dans les systèmes embarqués la mémoire attribuée à un problème spécifique était parfois limitée à une dizaine de bits.

Ces problèmes sont revenus subitement à la mode dans les années 2010 avec l'explosion des données. La motivation est que pour administrer de grands réseaux il est parfois préférable d'avoir des algorithmes robustes et rapides (quitte à ce qu'ils fassent des erreurs d'approximations) plutôt qu'on algorithme exact mais qui est tout de suite dépassé par la taille des données.

Un problème important dans ce domaine est le problème APPROXIMATECOUNTING : il s'agit tout simplement d'évaluer le nombre de paquets qui sont arrivés pendant un intervalle donné. Le but du projet est d'étudier trois algorithmes probabilistes extrêmement efficaces pour résoudre APPROXIMATE-COUNTING.

Motivations et position du problème APPROXIMATECOUNTING

Imaginons que l'on dispose d'un compteur binaire `C` qui peut utiliser n bits de mémoire, et l'on souhaite mettre à jour `C` à chaque nouvelle arrivée d'un paquet de données dans notre réseau. Le compteur `C` va alors successivement prendre les valeurs suivantes :

		1	2	3		...			n
	C $=$	0	0	0	0	0	0	0	0
Après 1 paquet	C $=$	0	0	0	0	0	0	0	1
Après 2 paquets	C $=$	0	0	0	0	0	0	1	0
Après 3 paquets	C $=$	0	0	0	0	0	0	1	1
			...						
Après $2^n - 1$ paquets	C $=$	1	1	1	1	1	1	1	1

Finalement, au bout de $m = 2^n - 1$ paquets la mémoire de n bits est saturée. Pour résoudre de façon parfaite le problème de comptage avec m paquets, il faut donc un espace de stockage de $\mathcal{O}(\log(m))$ bits. Nous allons analyser trois algorithmes différents qui utilisent astucieusement l'aléatoire pour diminuer spectaculairement cette borne de $\log_2(m)$:

- MISSEDCOUNT: Mémoire en $\log_2(m) - \mathcal{O}(1)$
- LOGCOUNT: Mémoire en $\mathcal{O}(\log(\log(m)))$
- MINCOUNT: Mémoire en $\mathcal{O}(1)$

En contrepartie ces trois algorithmes ne renvoient qu'une valeur approchée (et aléatoire) de m. Formellement, ces algorithmes produisent chacun une suite de variables aléatoires $(C_m)_{m\geq 1}$ telles que

- Le calcul permettant de passer de C_m à C_{m+1} lorsqu'un nouveau paquet arrive se fait très rapidement ;
- C_m est une estimation aléatoire de m qui est sans biais : on a $\mathbb{E}[C_m] = m$ pour tout m.

L'objectif de ce projet est de faire l'analyse théorique et expérimentale de ces trois algorithmes.

1 L'algorithme MISSEDCOUNT

L'algorithme MISSEDCOUNT est le plus rudimentaire des trois, on verra qu'il est bien moins performant que LOGCOUNT et MINCOUNT. L'idée est de "rater" (*to miss* en anglais, d'où le nom) volontairement certains paquets de données pour saturer moins vite la mémoire.

Formellement l'algorithme MISSEDCOUNT fonctionne de la façon suivante :

- On fixe un paramètre $K \geq 1$.
- Pour $m = 0$, on pose $Y_0 = 0$.
- Pour chaque $m \geq 1$ on tire une variable de Bernoulli B_m de moyenne $1/K$,
  - Si $B_m = 0$ on "rate" le paquet : on pose $Y_m = Y_{m-1}$.
  - Si $B_m = 1$ on compte le paquet : on pose $Y_m = Y_{m-1} + 1$.
  - On pose $C_m = K \times Y_m$.

Voici un exemple d'exécution de MISSEDCOUNT pour 5 paquets et $K = 3$:

m	B_m	Y_m	C_m
0	-	0	-
1	0	0	0
2	0	0	0
3	1	1	3
4	1	2	6
5	0	2	6

S1. Tracer sur le même graphique 15 trajectoires $(C_1, C_2, \ldots, C_m)$ pour $m = 1000$ et $K = 10$. Tracer également sur ce graphique la droite $m \mapsto m$. (Pour pouvoir comparer visuellement avec les deux autres algorithmes il est conseillé de fixer l'ordonnée maximale à $2m$, à l'aide de la commande de matplotlib : `plt.axis([0,m,0,2*m])`.)
(Vous devez constater que les trajectoires de (C_m) sont assez proches de la valeur exacte.)

S2. Tracer sur un autre graphique 15 trajectoires $(C_1, C_2, \ldots, C_m)$ pour $m = 1000$ et $K = 50$.
(Vous devez observer que les performances se dégradent : les courbes de (C_m) fluctuent beaucoup plus autour de $m \mapsto m$.)

T1. Quelle est la loi de C_m? En déduire que C_m est sans biais et calculer $\mathrm{Var}(C_m)$ en fonction de m et K.

Coût de stockage des Y_m

On s'intéresse maintenant à l'espace de stockage requis pour MISSEDCOUNT, on doit donc contrôler la taille de[1] Y_m.

T2. L'inégalité de Hoeffding[2] affirme que pour une variable Binomiale $\mathcal{B}$ de paramètre n, p, on a pour tout $x > 0$

$$\mathbb{P}(\mathcal{B} \geq np + x\sqrt{n}) \leq \exp(-2x^2).$$

En utilisant l'inégalité de Hoeffding, vérifier que si $K \leq 10$ alors on a

$$\mathbb{P}(Y_m \geq 2m/K) \leq \exp(-m/50).$$

Au maximum on a $Y_m = m$ donc on a besoin de $\log_2(m)$ bits pour stocker Y_m (comme dans le cas de l'algorithme naïf dans l'introduction) mais nous venons de montrer qu'avec grande probabilité Y_m est inférieure à $2m/K$. Ainsi en pratique on se limite à stocker Y_m avec $\log_2(m) - \log_2(K/2)$ bits et on économise quelques bits. Plus K est grand plus on économise de mémoire, mais la question T1 montre que l'erreur causée par l'aléa grandit également quand K grandit.

L'idée de LOGCOUNT est de pousser la stratégie de MISSEDCOUNT encore plus loin pour économiser beaucoup plus de mémoire.

2 L'algorithme LOGCOUNT

L'algorithme LOGCOUNT fonctionne de la façon suivante :

- Pour $m = 0$, on pose $C_0 = 0$, $Y_0 = 0$.
- Pour chaque $m \geq 1$ on tire une variable de Bernoulli B_m de moyenne $2^{-Y_{m-1}}$,
  - Si $B_m = 1$ on pose $Y_m = Y_{m-1} + 1$.
  - Si $B_m = 0$ on pose $Y_m = Y_{m-1}$.
  - On pose $C_m = 2^{Y_m} - 1$.

Voici un exemple d'exécution de LOGCOUNT pour 5 paquets :

[1] En effet, en pratique lors de l'implémentation de MISSEDCOUNT il est inutile de calculer C_m à chaque étape de l'algorithme. Seul Y_m est stocké en mémoire, on ne calcule C_m qu'à la toute fin.

[2] Voir par exemple : `https://fr.wikipedia.org/wiki/Inégalité_de_Hoeffding`

m	B_m	Y_m	C_m
0	-	0	0
1	1	1	1
2	1	2	3
3	0	2	3
4	0	2	3
5	1	3	7

Nous allons vérifier théoriquement et expérimentalement que C_m est sans biais, et évaluer l'espace mémoire requis par LOGCOUNT.

S3. Tracer sur le même graphique 15 trajectoires $(C_1, C_2, \ldots, C_m)$ pour $m = 10000$. Tracer également sur ce graphique la droite $m \mapsto m$. (Prendre à nouveau un axe des ordonnées limité à $2m$.)

T3. Calculer pour tout m la quantité

$$\mathbb{E}[C_{m+1} \mid C_m].$$

En déduire que pour tout $m \geq 1$ on a $\mathbb{E}[C_m] = m$.

Coût de stockage des Y_m

On cherche à évaluer l'espace de stockage requis pour LOGCOUNT en estimant Y_m.

T4. En utilisant une inégalité célèbre du Poly de MAP361, démontrer que $\mathbb{E}[Y_m] \leq \log_2(m+1)$. Avec une autre inégalité célèbre, en déduire que pour tout $A > 0$ on a $\mathbb{P}(Y_m > A\log_2(m+1)) \leq 1/A$.

Nous allons utiliser `python` pour obtenir une majoration bien meilleure de $\mathbb{P}(Y_m > A\log_2(m+1))$, pour A, m fixés.

T5. Pour $0 \leq k \leq m$ on note $p_k^{(m)} = \mathbb{P}(Y_m = k)$. Justifier que pour tout $m \geq 1$ on a

$$p_k^{(m)} = 2^{-(k-1)} p_{k-1}^{(m-1)} + (1 - 2^{-k}) p_k^{(m-1)} \qquad \text{pour tout } k \geq 1,$$
$$p_0^{(m)} = (1 - 2^{-k}) p_0^{(m-1)}.$$

S4. Déduire de la question T5 un code `python` qui calcule $p_k^{(m)}$. (Il n'est pas demandé une simulation mais un code qui calcule réellement de façon exacte $p_k^{(m)}$.) Utiliser votre code pour :

1. Tracer la distribution de probabilité $k \mapsto \mathbb{P}(Y_m = k)$ pour $m = 50$.
2. Calculer la probabilité $\mathbb{P}(Y_m \geq 2\log_2(m+1))$ pour $m = 50$.

(Pour vérifier votre code : je trouve $p_4^{(11)} = 0.34508....$)

On voit ainsi que l'on peut sans problème pour les applications pratiques supposer que Y_m sera bornée par $2\log_2(m+1)$, et donc qu'il est suffisant d'avoir un espace de stockage de $\mathcal{O}(\log_2(\log_2(m)))$ bits.

Performances de l'estimateur C_m

S5. On cherche maintenant à évaluer la probabilité que l'algorithme donne une estimation correcte de m à un facteur 2 près. Pour cela, tracer (en utilisant le code de la fonction S4)

$$m \mapsto \mathbb{P}\left(\frac{m}{2} \leq C_m \leq 2m\right)$$

pour $m \in \{1, 2, \ldots, 60\}$. *(Il est normal de trouver un graphique assez irrégulier. La performance de l'estimateur pour ce critère change beaucoup si m est une puissance de 2.)*

3 L'algorithme MINCOUNT

L'algorithme MINCOUNT est très différent, il utilise les propriétés de variables aléatoires continues uniformes. L'algorithme fonctionne de la façon suivante :

- Pour $m = 0$, on pose $V_0 = 1, W_0 = 1, C_0 = 1$.
- Pour chaque $m \geq 1$:
  - On tire une variable aléatoire uniforme U_m sur $[0, 1]$.
  - On définit V_m comme la plus petite valeur entre V_{m-1}, W_{m-1}, U_m. On définit W_m comme la deuxième plus petite valeur entre V_{m-1}, W_{m-1}, U_m.
  - On pose $C_m = 1/W_m$.

L'implémentation pratique de MINCOUNT requiert donc simplement de stocker deux nombres réels en mémoire (les nombres V_m et W_m), ce qui ne dépend pas de m et coûte 64 bits de mémoire[3] (par défaut un nombre réel est stocké sur 32 bits en python). Voici un exemple d'exécution de MINCOUNT pour 5 paquets de données :

m	U_m	V_m	W_m	C_m
0	-	1	1	1
1	0.31	0.31	1	1
2	0.91	0.31	0.91	1.0989
3	0.60	0.31	0.60	1.6667
4	0.77	0.31	0.60	1.6667
5	0.09	0.09	0.31	3.2258

S6. Tracer sur le même graphique 15 trajectoires $(C_1, C_2, \ldots, C_m)$ de MINCOUNT pour $m = 10000$. Tracer également sur ce graphique la droite $m \mapsto m$. (Prendre à nouveau un axe des ordonnées limité à $2m$.)

T6. Par construction la variable aléatoire W_m a exactement la loi de la deuxième plus petite valeur dans un échantillon de m uniformes indépendantes. Démontrer que la fonction de répartition de W_m a pour expression

$$\mathbb{P}(W_m \leq t) = 1 - (1-t)^m - nt(1-t)^{m-1} \qquad \text{(pour } t \in [0,1]).$$

En déduire la densité de f_m de W_m puis démontrer que $C_m = 1/W_m$ est sans biais.

S7. Il est possible de démontrer que lorsque $m \to +\infty$ on a $C_m/m \overset{\text{(loi)}}{\to} \mathcal{C}$, où $\mathcal{C}$ a pour densité $x \mapsto \exp(-1/x)\frac{1}{x^3}1_{0<x<+\infty}$. Vérifier par une simulation cette convergence en loi.
(À vous de choisir la simulation appropriée pour illustrer la convergence.)

4 Comparaison de MINCOUNT et LOGCOUNT : variance empirique

Les deux analyses que nous venons de faire pour MINCOUNT et LOGCOUNT ne permettent pas de conclure pour savoir quel est le meilleur algorithme. En effet les coûts de stockage respectifs de 64 bits et $\log_2(2\log_2(m+1))$ bits sont tous deux très faibles et ne sont pas un facteur limitant pour des machines actuelles.

[3] Il est en fait inexact de dire que MINCOUNT utilise une mémoire constante car pour pouvoir avoir une bonne précision sur C_m lorsque m augmente (et donc avoir un estimateur réellement non biaisé) il faut stocker W_m avec une précision de plus en plus grande. En pratique on se rend compte avec python que MINCOUNT utilise également une moyenne en $\log(\log(m))$, comme LOGCOUNT.

Pour réellement comparer les deux algorithmes il est plus pertinent de comparer les erreurs commises à cause de l'aléa de l'algorithme. Pour cela nous allons comparer les *variances empiriques* de chaque algorithme. La variance empirique d'un échantillon de S simulations $z_1, \dots, z_S$ est la quantité

$$\frac{1}{S}\sum_{s=1}^{S}(z_s)^2 - \left(\frac{1}{S}\sum_{s=1}^{S} z_s\right)^2.$$

S8. Pour chacun des deux algorithmes et pour $m \in \{5, 10, 15, 20\}$ calculer la variance empirique d'un échantillon de $S = 10000$ simulations de C_m. Afficher les 8 points obtenus sur un même graphique (en mettant m en abscisse). Conclure : quel semble-t-être le meilleur algorithme?

Conclusion historique

L'algorithme LOGCOUNT a été introduit en 1977 par Morris[4], alors chercheur à *Bell Labs*. Grâce à cet article Morris est considéré comme le pionnier des *stream algorithms*. La version de MINCOUNT que nous avons présentée est une version simplifiée d'un algorithme inventé en 1985 par Flajolet et Martin[5]. Il s'agissait en fait du problème plus difficile d'estimer le nombre de paquets **différents** arrivés dans le réseau pendant un intervalle donné, mais le principe est le même : utiliser les propriétés asymptotiques des plus petites valeurs parmi m uniformes. Cette approche a culminé dans les années 2000 avec l'algorithme HYPERLOGLOG[6], inventé par différents chercheurs travaillant à l'époque sur le plateau de Saclay. Parallèlement une équipe de chercheurs israéliens de *Microsoft Research*[7] introduisait des idées géométriques (géométrie convexe en grande dimension) pour ce même problème.

L'analyse des algorithmes probabilistes pour les flots de données est toujours un domaine très actif de recherche fondamentale et un problème crucial pour la recherche appliquée.

[4] R.Morris. Counting large numbers of events in small registers. *Communications of the ACM* 21, 10 (1977), 840–842.

[5] Flajolet, P., and Martin, G. N. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences* 31, 2 (1985), 182–209.

[6] Flajolet, P., Fusy, É., Gandouet, O., Meunier, F. (2007). Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In Discrete Mathematics and Theoretical Computer Science (pp. 137-156).

[7] Alon, N., Matias, Y., Szegedy, M. (1999). The space complexity of approximating the frequency moments. Journal of Computer and system sciences, 58(1), 137-147.