Vérification des files de priorité dans le modèle de streaming

Nathanaël François
Stage réalisé sous la direction de Frédéric Magniez

19 août 2011

Résumé

Le but de ce stage était d'étudier des problèmes dans le modèle de streaming. Dans ce modèle, une machine de Turing n'a jamais accès à toute l'entrée, mais seulement à un flux sur lequel sa tête de lecture ne peut qu'avancer. Copier en mémoire tout ce qui a été vu jusqu'ici ramènerait le problème au modèle classique, on s'intéresse donc uniquement à des algorithmes qui utilisent un espace sous-linéaire. Par ailleurs, on considère des algorithmes randomisés avec une probabilité d'erreur bornée, car les algorithmes déterministes nécessitent presque toujours une mémoire au moins linéaire dans ce modèle. Un modèle plus élaboré autorise plusieurs passages sur l'entrée.

Nous nous sommes principalement intéressés au cas de la vérification des files de priorité. Dans ce problème, on a en entrée l'historique de l'entrée de tâches de diverses priorités et de leur exécution, et cette entrée est valide si et seulement si à chaque fois qu'une tâche est exécutée, il s'agit de la tâche de plus haute priorité parmi celles en attente, et qu'à la fin toutes les tâches sont effectuées. Les files de priorité ont déjà été étudiées dans le modèle de streaming, notamment par Chakrabarti, Cormode, Kondapally et McGregor dans [CCKM10], mais on ne connaissait pas de borne fine sur la complexité en espace.

Ce problème présente des similarités avec celui des mots de Dyck à plusieurs parenthèses, étudié par Magniez, Mathieu et Nayak dans le modèle de streaming dans [MMN10]. Les résultats présentés ici sont analogues à ceux de [MMN10].

1 Introduction

Dans la partie 2, nous définissons formellement le modèle de streaming et les langages PrioritYQueue et Dyck[s].

Dans la partie 3 nous présentons un algorithme probabiliste dans le modèle de streaming en une passe qui décide si une file de priorité est valide (donc si elle appartient à PrioritYQueue) avec probabilité d'erreur au plus n^{-c} (c constante arbitraire) en espace $\tilde{O}(\sqrt{n})$. Cet algorithme est similaire à celui

de [CCKM10], mais nous utilisons une présentation différente. Nous donnons ensuite de bonnes raisons de penser qu'il n'est pas possible d'améliorer sa complexité, même en utilisant des timestamps [1].

En effet, dans la partie 4, on réduit PRIORITYQUEUE avec des timestamps à un problème de communication à $3n$ joueurs où les messages envoyés correspondent à l'état de la mémoire, puis à un problème de communication simple à 3 joueurs. On n'a cependant pas réussi à prouver la borne inférieure sur la complexité de communication de ce problème, donc ce n'est qu'une conjecture. Par ailleurs, on démontre également qu'une généralisation des problèmes DYCK[s], PRIORITYQUEUE et d'autres problèmes de ce type a une complexité en espace $\Omega(n)$ dans le modèle de streaming à une passe.

Dans la partie 5, on se pose la question d'un algorithme à deux passes. En effet, [MMN10] montre que la complexité en espace de DYCK[s] est $\Theta(\sqrt{n})$ dans le modèle de streaming probabiliste en une passe, et $O((\log n)^2)$ dans le modèle à deux passes. En adaptant les idées de cet article, on obtient un algorithme probabiliste dans le modèle de streaming en une passe pour PRIORITYQUEUE en espace $O((\log n)^2)$.

2 Définition et préliminaires

2.1 Le modèle de streaming

Le modèle de streaming considère des problèmes dont on ne peut pas charger l'entrée en mémoire, et où on ne peut y avoir qu'un accès séquentiel. En terme d'applications, ces algorithmes servent pour manipuler des données massives qui tiennent sur des disques durs mais pas dans une mémoire vive.

Définition 1. *Soit Σ un alphabet. Un algorithme de streaming en k passes A en espace $s(n)$ est un algorithme tel que pour toute entrée $x \in \Sigma^n$:*

- *A fait k passes séquentielles sur x.*
- *A utilise une mémoire de taille $s(n)$ bits.*

On dit que A est bidirectionnel si après avoir atteint la fin de l'entrée, il effectue la passe suivante en sens inverse. k est alors le nombre total de passes effectuées dans les deux sens.

Il existe d'autres modèles mais le nôtre convient dans le but de ce rapport. On ne s'intéresse qu'au modèle en une passe dans le rapport sauf dans la partie 5 où l'on considère le modèle à deux passes.

On remarque que si on peut recopier l'entrée en mémoire, alors on se ramène au modèle général d'un algorithme. Pour cette raison, un résultat dans le modèle de streaming est intéressant seulement s'il donne une complexité en espace sous-linéaire.

1. la borne inférieure sans timestamps peut s'obtenir par une simple réduction à DYCK[s] et est un résultat de [CCKM10]

2.2 Quelques problèmes

Un problème très simple dans le modèle de streaming en une passe est MULTISETEQUALITY. En effet on peut le résoudre en espace $O(\log n)$.

Définition 2. *Soient E, F deux multi-ensembles, dont les éléments sont données dans un ordre arbitraire.* $(E, F) \in$ MULTISETEQUALITY *si et seulement si $E = F$.*

Pour ce problème, on se donne $c \geq 1$ une constante, $p \in [n^{c+1}, 2n^{c+1}]$ un premier, $\alpha \in \{0, \ldots, p-1\}$ uniformément aléatoire. On rappelle le lemme de Schwartz-Zippel :

Lemme 1 (Schwartz-Zippel)**.** *Soit $P(X_1, \ldots, X_n)$ un polynôme non nul de degré d sur un corps fini F. Soit $r_1, \ldots, r_n$ des variables aléatoires uniformes et indépendantes dans F. Alors $Pr[P(r_1, \ldots, r_n) = 0] \leq \frac{d}{|F|}$.*

Si $E \neq F$, alors le polynôme de $\mathbb{Z}/p\mathbb{Z}$ $h(X) = \sum_{y \in E} X^y - \sum_{z \in F} X^z$ est non nul. En particulier, on a $h(\alpha) \neq 0$ avec probabilité au moins $1 - \frac{n}{n^{1+C}} \geq 1 - n^{-c}$. En particulier, choisir un α aléatoire et vérifier que le polynôme vaut 0 peut se faire en espace $O(\log n)$ dans le modèle de streaming en une passe : pour chaque élément x que l'on voit, on ajoute ou on retranche α^x à la valeur que l'on a déjà selon si x est dans E ou F.

Par la suite, on note $\text{hash}_+(h, \alpha, x) = h + \alpha^x \mod p$ et $\text{hash}_-(h, \alpha, x) = h - \alpha^x \mod p$, avec p comme plus haut. Ces fonctions permettent d'ajouter et de retirer des éléments à un hashcode et seront utiles dans les algorithmes des parties suivantes, selon le même principe que plus haut.

DYCK[s] est le langage des mots bien parenthésés avec s types de parenthèses.

Définition 3. *Soit s un entier positif. Alors* DYCK[s] *est le langage sur l'alphabet $\{a_1, \bar{a_1}, \ldots, a_s, \bar{a_s}\}$ défini par* $\text{DYCK[s]} = \epsilon + \sum_{i=1}^{s} a_i.\text{DYCK[s]}.\bar{a_i}.\text{DYCK[s]}$

Ce problème a été étudié dans le modèle de streaming par Magniez, Mathieu et Nayak dans [MMN10]. Nos résultats s'inspirent de cet article mais concernent un autre problème.

On peut également voir DYCK[s] comme le langage des piles "last in first out" sur un alphabet à s éléments.

Une file de priorité est une suite d'insertion et d'extraction telle que chaque extraction correspond au maximum des insertions précédentes qui n'ont pas encore été extraites.

En d'autres termes, à chaque fois qu'une tâche est traitée, il faut que ce soit la tâche de plus haute priorité parmi celles en attente.

Définition 4. *Soit* $\Sigma = \{1, \bar{1}, \ldots, N, \bar{N}\}$, $\sigma \in \Sigma^*$. $\sigma \in$ PriorityQueue *si et seulement si l'une de ces affirmations est vraie :*

- $\sigma = \epsilon$
- *Il existe* $u, w \in \Sigma^*$, $v \in \{1, \ldots, N\}^*$, $x \in \{1, \ldots, N\}$, $\sigma = uxv\bar{x}w$ *avec* $uvw \in$ PriorityQueue *et* x *la plus grande valeur de* σ.

On dit que σ *est une file de priorité bien formée.*

Ce problème a été étudié dans le cadre du modèle de streaming à une passe par Chakrabarti, Cormode, Kondapally et McGregor dans [CCKM10]. Cependant, l'algorithme que nous proposons dans la suite est plus simple (mais de même complexité) et nos résultats sur le modèle de streaming à deux passes sont nouveaux.

2.3 Quelques notions supplémentaires

On peut ajouter à ces deux problèmes des timestamps. Cela signifie qu'à chaque extraction on ajoute la date t de l'insertion à laquelle elle correspond dans la définition du langage. Ceci assure que les timestamps sont uniques puisque chaque extraction ne correspond qu'à une seule insertion. Les timestamps peuvent ne rendre le problème que plus facile puisqu'on dispose d'information supplémentaire.

Notamment, dans le modèle de streaming en une passe, Dyck[s] avec timestamps a une complexité en espace $O(\log n)$. En effet si on met tout dans un hashcode qui contient le type de parenthèse, la hauteur et le timestamp pour la parenthèse fermante, l'instant pour la parenthèse ouvrante, on garantit qu'à chaque parenthèse correspond une parenthèse de même type et de même hauteur. Il suffit ensuite de vérifier que la hauteur ne descend pas en dessous de 0 comme pour Dyck[1].

Enfin, pour ces problèmes, on définit un creux comme un sous-mot de la forme $\bar{x}y$, $x, y \in \Sigma$, i.e. une extraction immédiatement suivie d'une insertion. Dans le cas de PriorityQueue, on définit x comme la valeur du creux.

On note r le nombre de creux d'un mot.

Lorsque plusieurs insertions peuvent correspondre selon la définition de PriorityQueue à une même extraction, on considère la dernière de ces insertions afin d'avoir une définition unique.

3 Algorithme en une passe pour PriorityQueue

Notre algorithme est d'abord construit à partir d'un algorithme qui nécessite un espace $\tilde{O}(r)$, où r est le nombre de creux dans la file de priorité σ.

On utilise ensuite des fenêtres de taille $\sqrt{n}$ où on simplifie σ de manière à garantir qu'il y a au plus un changement de direction par fenêtre. On utilise

une mémoire supplémentaire $O(\sqrt{n})$ pour maintenir la fenêtre, et donc la complexité en mémoire de l'algorithme final est $\tilde{O}(\sqrt{n})$.

3.1 Idée de base : plusieurs instances de MultiSetEquality

Considérons une file de priorité de la forme $\sigma = x_1 x_2 \dots x_m \bar{y_1} \dots \bar{y_{m'}}$. Vérifier qu'un stream est une file de priorité de cette forme se ramène à vérifier l'égalité des multi-ensembles $\{x_1, \dots, x_m\}$ et $\{y_1, \dots, y_{m'}\}$ (donc en particulier $m = m'$ et à vérifier $y_1 \geq y_2 \geq \dots \geq y_{m'}$.

On peut donc le faire avec un seul hashcode comme pour MultiSetEquality et en comparant chaque élément extrait au précédent. Ceci prend un espace $O(\log n)$.

Cependant, on ne peut pas se contenter de cette méthode dans le cas général, i.e. lorsqu'il y a au moins un creux. En effet, $\sigma = 2.1.\bar{1}.4.\bar{4}.\bar{2}$ a bien les mêmes éléments insérés et extraits, et chaque suite d'extraction est décroissante, mais 1 est extrait avant que 2 ne le soit et alors qu'il a déjà été inséré.

On doit donc maintenir plusieurs valeurs en mémoire et pas seulement le hashcode de la différence entre les insertions et les extractions. La méthode la plus simple serait alors de retenir chaque insertion en mémoire jusqu'à rencontrer l'extraction correspondante, et de rejeter si on rencontre une extraction qui ne correspond pas à l'insertion maximale. Cette méthode nécessite cependant une mémoire linéaire.

Il faut donc grouper les insertions dans des hashcodes mais avoir un nombre de hashcodes suffisant pour distinguer les erreurs. L'idée est qu'une extraction ne va dans le bon hashcode que si elle est correcte. La figure 1 illustre un tel groupement, où on change de hashcode après chaque creux. Son problème est qu'il est difficile de retrouver le bon hashcode lorsqu'on arrive à l'extraction.

Cependant, on remarque qu'on doit déjà utiliser le nombre de creux, puisqu'après chacun d'eux on crée un nouveau hashcode. Comme on peut retenir les valeurs des creux, on aimerait pouvoir retenir avec les valeurs des insertions qui suivent le creux. C'est ce que nous permet le lemme 2 : pour toute file de priorité, il existe une file de priorité équivalente telle que, pour chaque insertion, la valeur (si elle existe) de l'extraction précédente (et donc du creux précédent, par définition) soit plus petite.

Lemme 2. *Soit σ une instance de* PriorityQueue. *Il existe σ' instance de* PriorityQueue *telle que :*

- *σ est bien formée si et seulement si σ' est bien formée*
- *Pour tout insertion $\sigma_\tau = u$ de σ, soit $\forall t \leq \tau, \sigma_t$ est une insertion, soit si $t_\tau = \max\{t \leq \tau | \sigma_t$ est une extraction$\}$ alors $\sigma_{t_\tau} < u$.*

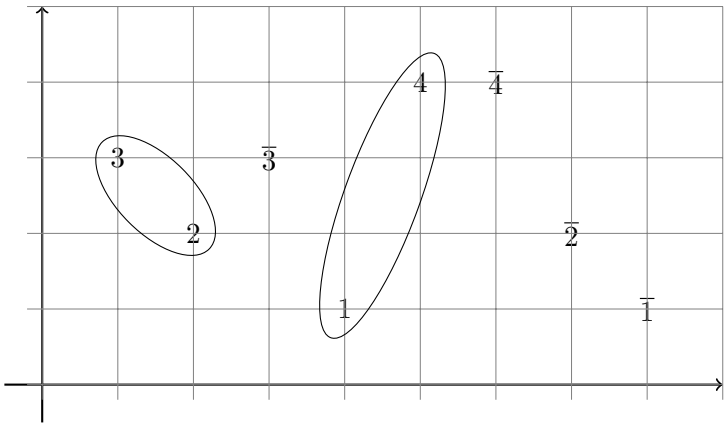

FIGURE 1 – Groupement permettant de vérifier la file de priorité avec deux hashcodes

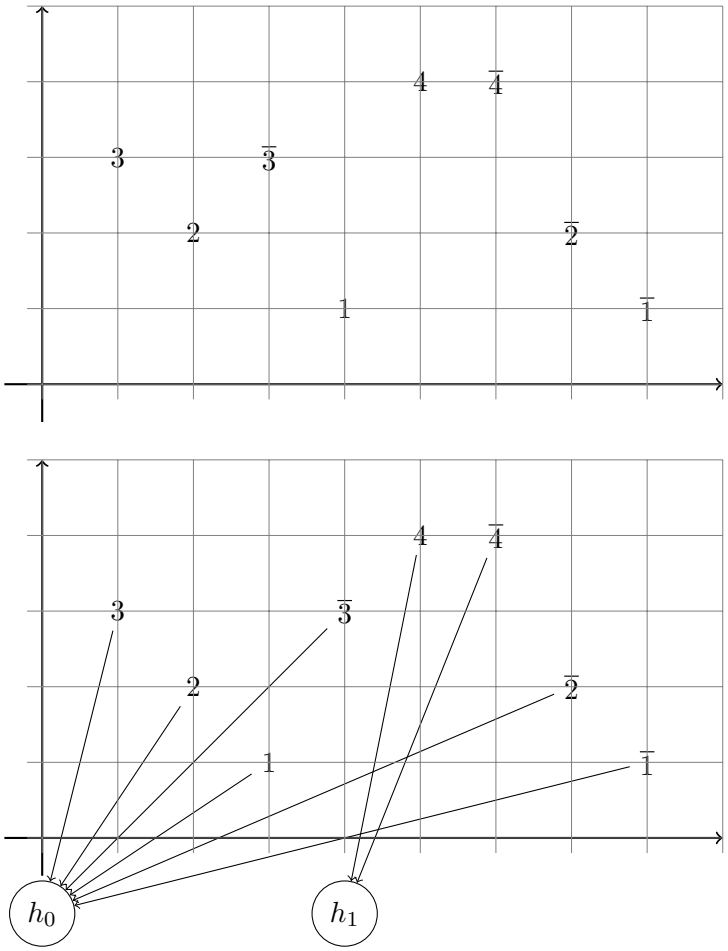

FIGURE 2 – Exemple de file de priorité modifiée selon le lemme 2 et de l'attibution des hashcodes

Dans la figure 2, on voit un exemple de l'application du lemme 2. Le 1 est inséré après l'extraction $\bar{3}$, donc comme $3 > 1$ on déplace le 1. C'est le seul changement effectué. 3, 2 et 1 vont dans h_0. $\bar{3}$ également car c'est le seul hashcode disponible. 4 va dans le nouveau hashcode du creux $\bar{3}$.4, à savoir h_1. $\bar{4}$ est plus grand que $\bar{3}$, donc il va dans h_1 (l'insertion correspondante n'a pas le droit d'être avant $\bar{3}$). $\bar{2}$ et $\bar{1}$ sont plus petits que $\bar{3}$ et vont donc dans h_0, car les insertions correspondantes ont été déplacées avant $\bar{3}$.

Soit σ' une telle file de priorité bien formée, τ et τ' tels que l'insertion $\sigma_\tau = u$ et l'extraction $\sigma_{\tau'} = \bar{u}$ se correspondent. Alors nécessairement, comme σ' est bien formée, pour tout $\tau < t < \tau'$, si σ_t est une extraction, $\sigma_t \geq u$. Donc on a $t_\tau = \max\{t \leq \tau | \sigma_t \text{ est une extraction}\} = \max\{t \leq \tau' | \sigma_t \text{ est une extraction}, \sigma_t < u\}$. Donc on arrive à caractériser t_τ uniquement en fonction de u et de τ' si on connaît les valeurs de tous les creux. Donc si on indice le hashcode dans lequel on met σ_τ par t_τ, on peut retrouver ce hashcode lorsqu'on arrive à $\sigma_{\tau'}$. En pratique, comme t_τ est forcément un creux, on préfère indicer par i si le i-ème creux est en t_τ.

Qui plus est, dans l'exemple précédent, si jamais σ' n'est pas bien formée et que σ_τ, $\sigma_{\tau'}$ et σ_{t_0} le prouvent, avec $\sigma_{t_0} = \bar{v}$, $v < u$ bien que $\tau < t_0 < \tau'$, dans ce cas on aura $\max\{t \leq \tau' | \sigma_t \text{ est une extraction}, \sigma_t < u\} \geq t_0 > t_\tau$. En particulier, on ne retrouvera pas le bon hashcode et l'algorithme échouera.

Il reste encore à prouver le lemme 2.

Démonstration. Soit σ une instance de PRIORITYQUEUE. On considère σ' l'instance obtenue en parcourant σ jusqu'à trouver $\sigma_\tau = u$ une insertion telle que $\sigma_{t_\tau} \geq u$. On pose alors $\forall t < t_\tau, \sigma'_t = \sigma_t, \sigma'_{t_\tau} = u, \forall t \in \{t_\tau+1, \ldots, \tau\}, \sigma'_t = \sigma_{t-1}$, $\forall t > \tau, \sigma'_t = \sigma_t$: on déplace σ_τ à l'instant t_τ en décalant le reste.

Comme la seule paire insertion-extraction qui est échangée est $(\sigma_\tau, \sigma_{t_\tau})$, la validité de σ ne change pas.

On répète cette étape jusqu'à ne plus trouver de σ_τ. Le processus termine car à chaque étape on fait passer une insertion derrière une extraction. À la fin, on a un σ' qui satisfait les deux propriétés du lemme. □

3.2 Première étape : algorithme en espace $\tilde{O}(r)$

Comme expliqué plus haut, le principe de l'algorithme 1 est de maintenir plusieurs hashcodes h_j contenant les valeurs insérées et extraites, de telle manière que lorsqu'on rencontre une extraction, elle soit mise dans le hashcode qui contient l'insertion correspondante seulement si elle est à un endroit légal.

Algorithme 1 Algorithme en une passe en espace $\tilde{O}(r)$

$i \leftarrow 0$; $m_0 \leftarrow -\infty$; $h_0 \leftarrow 0$; $\forall x, C_x \leftarrow 0$; $t \leftarrow 0$
α random integer in $\{1, \ldots, p-1\}$
while $t < |\sigma|$ **do**
  $t \leftarrow t + 1$
  if σ_t is insert **then**
    ReadInsert(σ_t)
  else
    if σ_{t-1} is insert **then**
      $i \leftarrow i + 1$; $m_i \leftarrow +\infty$; $h_i \leftarrow 0$
    end if
    ReadExtract(σ_t)
  end if
end while
FinalCheck

Algorithme 2 ReadInsert

$k \leftarrow \max\{j | m_j < \sigma_t, j \leq i\}$
$h_k \leftarrow \text{hash}_+(h_k, \alpha, \sigma_t)$
if $\exists j < i, m_j = \sigma_t$ **then**
  $C_{\sigma_t} \leftarrow C_{\sigma_t} + 1$
end if

Algorithme 3 ReadExtract

$k \leftarrow \max\{j | m_j < \sigma_t, j \leq i\}$
$h_k \leftarrow \text{hash}_-(h_k, \alpha, \sigma_t)$
Check$(m_i \geq \sigma_t)$
$m_i \leftarrow \sigma_t$
if $\exists j < i, m_j = \sigma_t$ and $C_{\sigma_t} > 0$ **then**
  $C_{\sigma_t} \leftarrow C_{\sigma_t} - 1$
end if

Algorithme 4 FinalCheck

for $j = 0$ to i **do**
  Check$(h_j = 0)$
end for
for $x \in \Sigma$ **do**
  Check$(C_x = 0)$
end for
Accept

Pour cela, le hashcode dans lequel est mis une valeur correspond toujours au hashcode de l'indice du plus petit bloc dans lequel il était possible qu'elle soit insérée puis immédiatement extraite sans que cela ne modifie la correction de la file de priorité.

Le paramètre C_x sert à détecter des erreurs dans le cas où il existe des répétitions, i.e. on a plusieurs fois la même insertion. C_x s'assure que tout suffixe de σ contient au moins autant de $\bar{x}$ que de x en comptant les premiers négativement et les sexonds positivement. Par ailleurs, il n'est nécessaire de le maintenir que pour les valeurs des creux, ailleurs on détecte déjà les errreurs. On peut l'ignorer pour comprendre l'esprit de l'algorithme et considérer à la place que les répétitions sont interdites.

Lemme 3. *Soit σ une file de priorité, $\sigma_\tau = x$ une insertion et $\sigma_{\tau'} = \bar{x}$ l'extraction correspondante. Soient h_k et h'_k les hashcodes dans lesquels σ_τ et $\sigma_{\tau'}$ sont respectivement comptés par l'algorithme 1.*

Alors $k = k'$ si et seulement si pour tout $\tau < t < \tau'$, si σ_t est une extraction, $\sigma_t \geq x$.

Démonstration. Supposons que les conditions sont satisfaites. À l'instant τ, x est ajouté à h_k, où $k = \max\{j | m_j < x, j \leq i\}$. À l'instant τ', soit i' l'indice du bloc courant. x est retranché de $h_{k'}$, où $k' = \max\{j | m_j < x, j \leq i'\}$. Or par hypothèse un élément inférieur à x ne peut être extrait après τ et avant τ'. Donc a fortiori, pour tout $j \in \{i+1, \ldots, i'-1\}$, $m_j \geq x$, et de plus à l'instant τ' on a $m'_i \geq x$, sinon l'algorithme rejette. Donc $k = k'$.

Réciproquement, supposons qu'il existe un t qui ne satisfait pas les conditions. Soit i l'indice du bloc courant à l'instant τ, i' à l'instant t et i'' à l'instant τ'. Comme on change d'indice entre une insertion et une extraction, on a nécessairement $i < i'$. Or $m'_i \leq \sigma_t < x$. Donc à l'instant τ, on va avoir $k' = \max\{j | m_j < x, j \leq i'\} \geq i' > i$. Par définition, $k \leq i$. Donc $k < k'$. □

Proposition 4. *L'algorithme 1 est un algorithme randomisé dans le modèle de streaming en une passe pour* PrioritYQueue *en espace $\tilde{O}(r)$. Si l'entrée est une file de priorité bien formée alors l'algorithme accepte toujours, sinon il refuse avec probabilité au moins $1 - n^{-c}$.*

Démonstration. Supposons $\sigma \in$ PrioritYQueue. Alors d'après le lemme 3, toute extraction est dans le même h_k que l'insertion correspondante. Donc en particulier, à la fin tous les h_k valent 0. De plus, soit $\sigma_\tau = x$ tel qu'à l'instant τ, si i est l'indice du bloc courant, il existe $j < i$ tel que $m_j = x$. Alors si i' est l'indice du bloc courant à l'instant où l'algorithme rencontre l'extraction correspondante, $j < i'$, donc C_x n'augmente pas globalement. On a donc la complétude.

Pour prouver la correction, supposons d'abord que les hashcodes sont sans collisions, i.e. qu'ils sont évalués à 0 si et seulement les valeurs ajoutées et soustraites sont les mêmes. En réalité, la probabilité qu'un h_j donné soit

évalué à 0 si ce n'est pas le cas est au plus n^{-c}, donc si on prouve que l'algorithme rejette sous notre hypothèse, cela signifie qu'il rejette avec probabilité au moins $1 - n^{-c}$ en réalité.

Soit σ une file de priorité telle les extractions ne soient pas égales aux insertions, i.e. $\sigma \notin$ MultiSetEquality. Alors nécessairement la somme des h_j est non nulle et par conséquent au moins un h_j est non nul. Donc l'algorithme rejette.

Soit $\sigma \in$ MultiSetEquality une file de priorité telle qu'une extraction $\sigma_\tau = \bar{x}$ ne corresponde à aucune insertion σ_t la précédant. Il se peut que σ_τ soit précédé d'insertions u de même valeur, mais celles-ci sont par hypothèse extraites avant. Donc à l'instant τ, nécessairement $C_x = 0$ par la preuve de la complétude plus haut. Soit i l'indice du bloc courant. Si avant que le bloc ne change, il y a une extraction de $y \neq x$ (donc $\bar{y} < \bar{x}$ car sinon on détecte une erreur), nécessairement $m_i < x$. Donc pour tout $t \leq \tau$, si $\sigma_t \in \{x, \bar{x}\}$, alors σ_t est compté dans un h_k avec $k < i$, et pour tout $t > \tau$, si $\sigma_t \in \{x, \bar{x}\}$, σ_t est compté dans un h'_k avec $k' \geq i$. Comme on a au moins un $\bar{x}$ de plus dans le premier ensemble, il existe j tel que $h_j \neq 0$. Si à l'inverse, toutes les extractions qui suivent σ_τ dans le même bloc valent également $\bar{x}$, alors on a $m_i = x$ pour le reste de l'algorithme. Mais la suite de σ contient au moins un x de plus de que $\bar{x}$, par hypothèse, donc à la fin de l'algorithme on a $C_x > 0$, et on détecte encore l'erreur.

Il nous reste le dernier cas à traiter : chaque extraction de σ est précédée par une insertion qui lui correspond, mais σ contient $\sigma_\tau = x$, $\sigma_t = \bar{y}$, $\sigma_{\tau'} = \bar{x}$ tels que σ_τ et $\sigma_{\tau'}$ se correspondent et $x > y$. Par le lemme 3, σ_τ est compté dans h_k et $\sigma_{\tau'}$ dans h'_k avec $k < k'$, donc à la fin nécessairement $h_k \neq 0$.

Comme on maintient r hashcodes de taille logarithmique, et jusqu'à r compteurs C_x non nuls (il n'est évidemment pas nécessaire de maintenir ceux dont la valeur est 0), on a un algorithme en espace $\tilde{O}(r)$. □

3.3 Algorithme en espace $\tilde{O}(\sqrt{n})$

Pour passer à un algorithme plus efficace, l'idée naturelle est de réduire r par un pré-traitement de la file qui ne change pas sa correction. Cependant, on ne peut pas a priori faire un tel pré-traitement dans le modèle de streaming en une passe.

Pour cela, on a recours à une fenêtre de taille $\sqrt{n}$ en mémoire à l'intérieur de laquelle on effectue le pré-traitement, et que l'on considère comme un seul bloc. On garantit qu'il n'y a qu'un creux par fenêtre, donc $r = \sqrt{n}$ aussi, et on a un algorithme en $\tilde{O}(\sqrt{n})$.

L est une liste ordonnée des insertions de la fenêtre auxquelles ne correspond encore aucune extraction. En maintenant L et en comparant son maximum avec chaque nouvelle sortie, on s'assure qu'on n'a pas à l'intérieur de la fenêtre de paire $(x, \bar{y})$ sans $\bar{y}$ entre les deux et avec $x > y$. On dé-

Algorithme 5 Algorithme en une passe en espace $\tilde{O}(\sqrt{n})$

$i \leftarrow 0$; $m_0 \leftarrow -\infty$; $h_0 \leftarrow 0$; $\forall x, C_x \leftarrow 0$; $L \leftarrow []$; $t \leftarrow 0$
α random integer in $\{1, \ldots, p-1\}$
while $t < |\sigma|$ **do**
  $t \leftarrow t + 1$
  if σ_t is insert **then**
    ReadInsert(σ_t)
    $L \leftarrow L \cup \{\sigma_t\}$
    if Size$(L) > \sqrt{n}$ **then**
      $i \leftarrow i + 1$; $m_i \leftarrow +\infty$; $h_i \leftarrow 0$
      for $u \in \sigma$ **do**
        if $\forall j \leq i, m_j \neq u$ and $C_u > 0$ **then**
          Reject
        end if
      end for
    end if
  else
    Check$(\sigma_t \geq \max(L))$
    if $\sigma_t > max(L)$ **then**
      ReadExtract(σ_t)
    else
      ReadExtract(σ_t)
      $L \leftarrow L - \{\sigma_t\}$
    end if
  end if
end while
FinalCheck

cide que la fenêtre est terminée quand L contient $\sqrt{n}$ éléments, ce qui peut faire des fenêtres plus grandes grâce aux simplifications des paires insertions/extractions à l'intérieur de la fenêtre.

Le fait de ne jamais changer de bloc à l'intérieur de la fenêtre revient à faire comme si toute la fenêtre était une suite d'extractions puis une suite d'insertions, avec au milieu les insertions qui sont immédiatement extraites dans cette fenêtre (et donc ne génèrent pas de hashcode). La figure 3 montre un exemple de cette réorganisation sur une fenêtre à l'intérieur d'une file de priorité.

On remarque que le lemme 3 s'applique encore à cet algorithme, grâce à la remarque plus haut sur le rôle de L (la preuve est la même que dans la sous-partie 3.2 si x et $\bar{y}$ ne sont pas dans la même fenêtre).

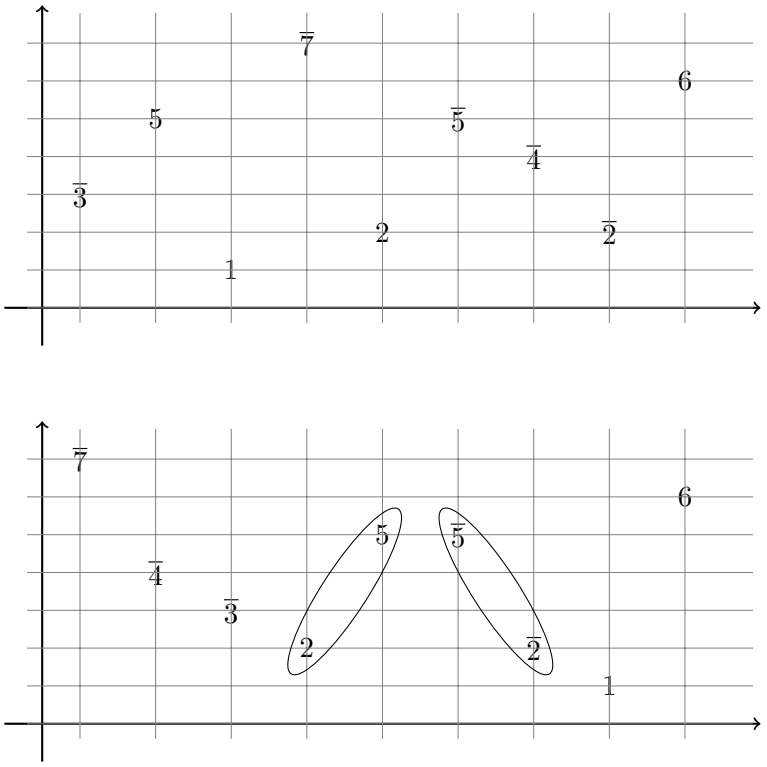

FIGURE 3 – Réorganisation de la file de priorité à l'intérieur d'une fenêtre sans changer les erreurs non internes à la fenêtre. Les parties entourées se correspondent et peuvent être ignorées.

Théoreme 5. *L'algorithme 5 est un algorithme randomisé dans le modèle de streaming en une passe pour* PRIORITYQUEUE *en espace* $\tilde{O}(\sqrt{n})$. *Si l'entrée est une file de priorité bien formée alors l'algorithme accepte toujours, sinon il refuse avec probabilité au moins* $1 - n^{-c}$.

Démonstration. La preuve de la complétude est identique à celle de la proposition 4 car on peut encore applique le lemme 3.

Prouvons donc la correction. Encore une fois, on considère d'abord que les hashcodes sont parfaits et ne s'évaluent à 0 que si les valeurs ajoutées et soustraites sont les mêmes, et donc lorsqu'on dira que l'algorithme rejette cela signifie qu'il rejette avec probabilité au moins $1 - n^{-c}$.

Si les insertions et extractions de σ ne sont pas identiques, la somme des hashcodes doit être non nulle donc au moins un h_j est non nul.

Si les insertions et extractions sont les mêmes, mais qu'il existe une extraction $\sigma_\tau = \bar{x}$ qui est précédée d'au moins autant de $\bar{x}$ que de x, alors la même preuve s'applique encore, à condition de considérer la dernière telle extraction. Soit i l'indice du bloc à l'instant τ. Soit $\sigma_{\tau'} = x$ l'insertion x suivante. À l'instant τ', si $m_i < x$, alors il y aura un $\bar{x}$ de plus que de x dans

les h_j avec $j < i$, donc un des h_j sera non nul. Si $m_i = x$ à l'instant τ', alors $C_x > 0$. Tout $\bar{x}$ qui pourrait redescendre la valeur de C_x est précédé par au moins un x de plus (par hypothèse, on a pris la dernière extraction fautive), et donc si C_x est décrémentée lorsque l'algorithme arrive à cette extraction, c'est qu'on a encore $m_i = x$, et comme m_i ne peut que décroître, c'était aussi le cas lorsque l'algorithme a rencontré l'insertion précédente, et C_x a été incrémentée. On a donc $C_x > 0$ à la fin de l'algorithme, et on rejette.

Enfin, supposons que chaque extraction est précédée par une extraction qui lui correspond, mais que σ contient $\sigma_\tau = x$, $\sigma_t = \bar{y}$, $\sigma_{\tau'} = \bar{x}$ tels que σ_τ et $\sigma_{\tau'}$ se correspondent et $x > y$. La preuve est exactement la même que pour la proposition 4, comme on peut toujours appliquer le lemme 3.

En prenant en compte la probabilité d'avoir un hashcode à 0 quand les valeurs ajoutées et soustraites ne sont pas les mêmes, on a une probabilité au moins $1 - n^{-c}$ de rejeter quand σ n'est pas une file de priorité bien formée.

Par construction de l'algorithme, on ne retient en mémoire que au plus $\sqrt{n}$ hashcodes de taille $O(\log n)$ et $2\sqrt{n}$ compteurs, en plus de T, qui est de taille $\sqrt{n}$. Donc l'algorithme est en espace $\tilde{O}(\log n)$. □

4 Bornes inférieures

4.1 Ébauche de preuve de borne inférieure pour PriorityQueue avec des timestamps

Chakrabarti, Cormode, Kondapally et McGregor montrent dans [CCKM10] que le résultat du théorème 5 est optimal, i.e. tout algorithme de streaming en une passe pour PriorityQueue nécessite un espace $\Omega(\log n)$, comme pour le problème Dyck[s]. Par ailleurs, on sait que les timestamps rendent le problème Dyck[s] trivial, comme montré dans la partie 2.3. Ce problème peut se résoudre dans le modèle de streaming en une passe en espace $O(\log n)$.

Cependant, il semblerait qu'il n'en soit pas de même pour PriorityQueue.

On considère le problème de communication à trois joueurs suivants :

Soit $X \subset \{q+1, \ldots, q+2m\}$ de taille m, Y, Z une partition de X. Alice connaît X, Bob connaît Y et Charlie connaît Z. Alice envoie un message à Bob qui envoie lui-même un message à Charlie. Charlie doit ensuite déterminer si $Y > Z$, i.e. si $\forall y \in Y, \forall z \in Z, y > z$.

On note M_A et M_B les messages d'Alice et de Bob respectivement, μ la distribution uniforme sur les instances **valides** de (X, Y, Z).

Lemme 6. *S'il existe un algorithme randomisé de streaming à une passe pour* PriorityQueue *avec des timestamps en espace $o(\sqrt{n})$ et une probabilité d'erreur au plus ϵ, alors il existe un protocole de communication avec aléa public pour le problème précédent tel que $|M_A| = o(m)$ et $I_\mu(M_B : Y|Z) = o(1)$ et une probabilité d'erreur au plus ϵ.*

Démonstration. On se donne l'instance suivante de PRIORITYQUEUE (avec timestamps), comme sur la figure 4 :

Soit $n = 2m^2$, $X_k \subset \{2mk+1, \ldots, 2m(k+1)\}$ de taille m et $Y_k \subset X_k$ pour $1 \leq k \leq m$. σ est composé des éléments de X_1 insérés dans un ordre arbitraire, puis des éléments de Y_1 extraits par ordre décroissant, puis de même avec tous les X_k et les Y_k. Ensuite, les éléments des $X_k - Y_k$ sont extraits par ordre décroissants en commençant par $k = m$.

$\sigma \in$ PRIORITYQUEUE si et seulement si pour tout k, $Y_k > Z_k = X_k - Y_k$.

On considère maintenant que cette instance est un problème de communication à $3m$ joueurs comme celui décrit plus haut, où A_1 envoie un message à B_1 qui envoie un message à A_2 et ainsi de suite jusqu'à B_m, qui envoie un message C_m qui envoie lui-même un message aux C_k suivants jusqu'à C_1.

Si il existe un algorithme de streaming en une passe qui utilise une mémoire $s(n)$ pour le problème PRIORITYQUEUE, alors il existe un protocole de communication pour le problème à $3m$ joueurs où la taille des messages n'excède jamais $s(n)$: il suffit aux joueurs d'appliquer l'algorithme de streaming et de transmettre l'état de la mémoire au joueur suivant. Si X_k, Y_k, Z_k sont valides, comme A_k décide arbitrairement de l'ordre des X_k, elle peut en particulier les mettre par ordre croissants et comme par hypothèse $Y_k > Z_k$, B_k et C_k peuvent générer des timestamps. Si ils ne sont pas valides, ils ne généreront pas de timestamps valides mais ça n'est pas un problème puisqu'ils doivent justement rejeter.

Il nous reste à revenir au cas à 3 joueurs. Pour cela, soit P le protocole obtenu à partir de l'algorithme de streaming, $1 \leq j \leq m$. On considère P_j le protocole suivant :

- Alice, Bob et Charlie utilisent l'aléa partagé pour créer des valeurs aléatoires de distribution $\mu^{\otimes(m-1)}$ des X_k, Y_k, Z_k pour $k \neq j$.
- Alice prend $X = X_j$, Bob $Y = Y_j$, Charlie $Z = Z_j$.
- Alice simule le protocole jusqu'à A_j, puis envoie le message M_{A_j} à Bob.
- Bob simule le protocole jusqu'à B_m, puis envoie le message M_{B_m} à Charlie.
- Charlie simule le protocole jusqu'à la fin, et accepte ou rejette en conséquence

Comme tous les autres (X_k, Y_k, Z_k) satisfont les conditions (ils sont pris selon la distribution μ), ce protocole accepte si et seulement si (X_j, Y_j, Z_j) satisfont les conditions. Donc P_j est un protocole pour le jeu à trois joueurs.

Le protocole P_j vérifie $|M_A^{(j)}| = |M_{A_j}| = o(m)$. Par ailleurs, on a $I_{\mu^{\otimes m}}(M_{B_m} : Y_1, \ldots, Y_m | Z_1, \ldots, Z_m) = o(m)$ car $|M_{B_m}| = o(m)$. Donc en particulier on a $\sum_{j=1}^{m} I_{\mu^{\otimes m}}(M_{B_m} : Y_j | Z_1, \ldots, Z_m, Y_1, \ldots, Y_{j-1}) = o(m)$. Or Y_j est indépendant, de Y_k et Z_k pour $k \neq j$. Donc en particulier, comme $I(a : b|c) \leq I(a : b)$ si a et b sont indépendants, on a $\sum_{j=1}^{m} I_{\mu^{\otimes m}}(M_{B_m} : Y_j | Z_j) = o(m)$. Donc $E_j[I_\mu(M_B^{(j)} : Y|Z)] = o(1)$. Donc il existe un j tel que le protocole P_j vérifie les bornes du lemme. □

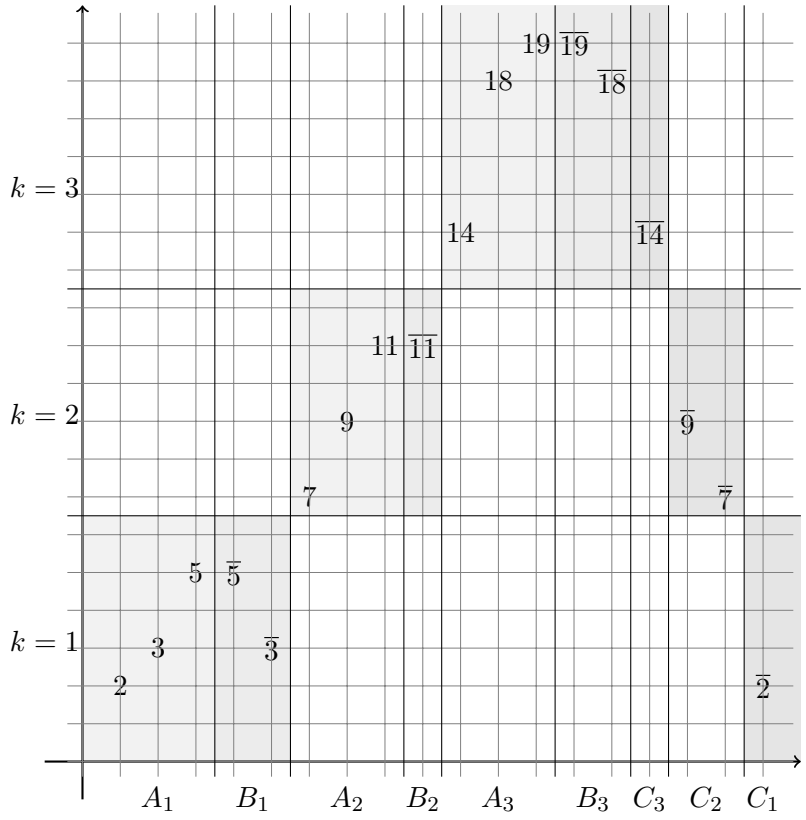

FIGURE 4 – Réduction du problème de communication à $3m$ joueurs à PRIORITYQUEUE, pour $m = 3$.

Malheureusement, nous n'avons pas de borne inférieure pertinente sur ce problème de communication à trois joueurs. Même en modifiant le problème pour qu'Alice envoie son message directement à Charlie sans que Bob ne puisse le lire, nous avons rencontré des difficultés dans la preuve. Ils nous paraît cependant une conjecture raisonnable que ce problème est bien difficile, et donc que tout algorithme de streaming pour PRIORITYQUEUE avec timestamps nécessite un espace $\Omega(\log n)$.

Ce qui justifie notre conjecture est que dans le jeu à trois joueurs, on a tendance à dire qu'Alice doit envoyer tout X à Bob pour qu'il vérifie que les éléments de Y sont maximaux (soit m bits d'information au moins), ou alors Bob doit envoyer le minimum de Y à Charlie (soit 1 bit d'information au moins).

4.2 Une borne inférieure pour le problème générique

On est tenté de généraliser les langages DYCK[S] et PRIORITYQUEUE au langage suivant, que l'on peut appeler BAG : on se donne une suite d'inser-

tions et d'extractions. C'est un élément du langage si et seulement si chaque extraction correspond à une insertion précédente.

En d'autres termes, Bag est PriorityQueue sans les priorités.

Définition 5. *Soit* $\Sigma = \{1, \bar{1}, \ldots, N, \bar{N}\}$, $\sigma \in \Sigma^*$. $\sigma \in$ Bag *si et seulement si l'une de ces affirmations est vraie :*

- $\sigma = \epsilon$
- *Il existe* $u, v, w \in \Sigma^*$, $x \in \{1, \ldots, N\}$, $\sigma = uxv\bar{x}w$ *et* $uvw \in$ PriorityQueue.

En s'inspirant des résultats sur Dyck[s] et PriorityQueue, on aurait tendance à penser qu'il existe un algorithme de streaming en une passe en espace $\tilde{O}(\sqrt{n})$ pour Bag.

En réalité, ce n'est pas le cas, et on a même une borne inférieure de la mémoire nécessaire dans ce modèle à $\Omega(n)$.

Proposition 7. *Soit* A *un algorithme randomisé de streaming à une passe pour* Bag *avec probabilité d'erreur au plus* $\frac{1}{3}$. *Alors* A *est en espace* $\Omega(n)$.

Démonstration. On réduit au problème SetDisjointness, dont la complexité en espace $\Omega(n)$ est connue.

En effet, considérons, pour $X, Y \subset I = \{1, \ldots, 3m\}$ de tailles m, l'instance suivante de Bag : les éléments de $I - X$ insérés dans un ordre arbitraire, puis les éléments de Y extraits dans un ordre arbitraire, puis les éléments de Y insérés dans un ordre arbitraire, puis les éléments de $I - X$ extraits dans un ordre arbitraire.

Toutes les extractions correspondent à une insertion, mais cette instance n'est dans Bag que si $Y \subset I - X$, en d'autres termes Y et X sont disjoints. □

5 Algorithme à deux passes pour PriorityQueue

Comme dans [MMN10], notre algorithme fonctionne en faisant une passe dans chaque sens avec compression des blocs à chaque fois. De l'information est perdue et il se peut qu'une erreur ne soit pas détectée à un passage, mais on s'assure qu'elle l'est alors dans l'autre sens.

Tout d'abord, nous avons besoin d'un algorithme en une passe de la droite vers la gauche, car contrairement à Dyck[s], PriorityQueue n'est pas symétrique. Il n'est pas nécessaire d'avoir un algorithme qui est déjà efficace en place comme l'algorithme 5 car de toute façon la compression interviendra plus tard.

5.1 Algorithme en une passe de la droite vers la gauche

En fait, l'algorithme de la droite vers la gauche est encore plus simple que le premier. En effet, dans le premier algorithme il était nécessaire de

déplacer les insertions pour qu'on puisse les retrouver au moment de l'extraction correspondante. Ici, comme ce sont les extractions qui sont traitées en premier, et que les contraintes de PriorityQueue portent justement sur les extractions, il n'est pas nécessaire de les déplacer : les contraintes permettent déjà de les retrouver.

Par exemple, dans la figure 5, 3 et 4 sont au-dessus de $\overline{3}$, qui est le minimum du bloc d'indice 1. Donc leur extraction est nécessairement dans ce bloc car on extrait toujours les plus grandes valeurs en premier. En revanche, 1 est en dessous de $\overline{3}$, donc son extraction doit être dans le bloc d'indice 0.

La gestion des répétitions est en revanche un petit peu plus complexe, et donc le fonctionnement des C_i dans cet algorithme est sensiblement différent de celui des C_x dans l'autre (notamment, il n'est pas nécessaire de vérifier qu'ils valent 0 à la fin) : ici pour chaque creux le C_i correspondant compte le nombre de répétitions de la valeur du creux dans le bloc d'indice i dont on a l'extraction mais pas l'insertion. Une fois qu'on a toutes les insertions, les insertions suivantes de cette valeur correspondent à une extraction du bloc précédent. Ce sont des détails que l'ont peut ignorer si on veut juste comprendre le fonctionnement général de l'algorithme. Il suffit pour cela de supposer qu'il n'y a pas de répétitions et que les conditions sur les C_i sont toujours satisfaites.

La vérification a la fin est plus simple dans cette version (à cause de la différence de comportement des C_i) et dont nous l'avons incluse dans l'algorithme principal.

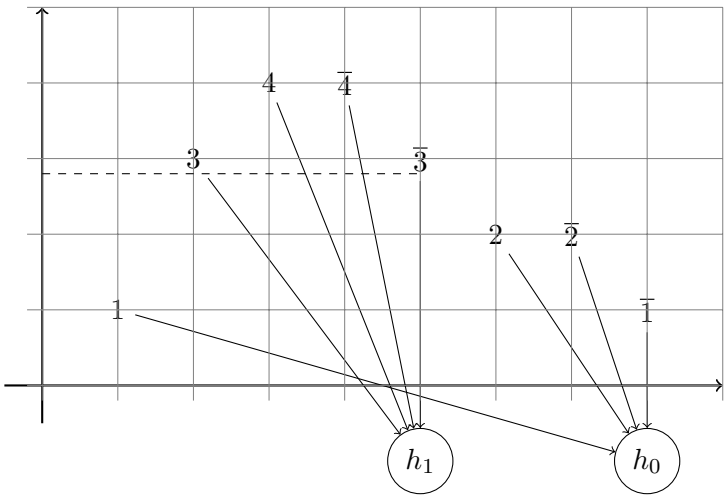

Figure 5 – Pour une passe de droite à gauche, il n'est pas nécessaire de déplacer les extractions pour retrouver les hashcodes

Algorithme 6 Algorithme en une passe de la droite vers la gauche en espace $\tilde{O}(r)$

$i \leftarrow 0$; $m_0 \leftarrow -\infty$; $h_0 \leftarrow 0$; $C_0 \leftarrow 0$; $t \leftarrow n$
α random integer in $\{1, \ldots, p-1\}$
while $t > 0$ **do**
  if σ_t is insert **then**
    ReadInsertReverse(σ_t)
  else
    if σ_{t-1} is insert **then**
      $i \leftarrow i + 1$; $m_i \leftarrow \sigma_t$; $h_i \leftarrow 0$; $C_i \leftarrow 0$
    else
      Check$(\sigma_t \geq \sigma_{t-1})$
    end if
    ReadExtractReverse(σ_t)
  end if
  $t \leftarrow t + 1$
end while
for $j = 0$ to i **do**
  Check$(h_j = 0)$
end for
Accept

Algorithme 7 ReadExtractReverse

$h_i \leftarrow \text{hash}_+(h_i, \alpha, \sigma_t)$
if $m_i = \sigma_t$ **then**
  $C_i \leftarrow C_i + 1$
end if

Algorithme 8 ReadInsertReverse

$k \leftarrow \max\{j \leq i | m_j < \sigma_t \text{ or } (m_j = \sigma_t \text{ and } C_j > 0\}$
$h_k \leftarrow \text{hash}_-(h_k, \alpha, \sigma_t)$
if $m_j = \sigma_t$ **then**
  $C_j \leftarrow C_j - 1$
end if

Commençons par un lemme dans l'esprit du lemme 3 :

Lemme 8. *Soit σ une file de priorité, $\sigma_\tau = \bar{x}$ une extraction et $\sigma_{\tau'} = x$ l'insertion correspondante. Soient h_k et h'_k les hashcodes dans lesquels σ_τ et $\sigma_{\tau'}$ sont respectivement comptés par l'algorithme 6.*

Alors $k = k'$ si et seulement si pour tout $\tau' < t < \tau$ tel que σ_t est une extraction, alors $\sigma_t \geq x$.

Démonstration. Supposons que les conditions sont satisfaites. À l'instant

τ, $\bar{x}$ est ajouté à h_i. Comme m_i a la valeur de la première extraction du bloc i et qu'on vérifie que les extractions sont croissantes, nécessairement $\bar{x} \geq m_i$. À l'instant τ', soit i' l'indice du bloc courant. Par hypothèse, quel que soit $t \in \{\tau', \ldots, \tau\}$, si σ_t est une extraction, $\sigma_t \geq x$. Donc pour tout $j \in \{i+1, \ldots, i'\}$, on a $m_j \geq x$. Supposons pour l'instant qu'on n'a pas $C_j > 0$ et $m_j = x$, et que si $m_i = x$ alors $C_i > 0$. Dans ce cas, nécessairement $k' = \max\{j \leq i' | m_j < \sigma_t \text{ or } (m_j = \sigma_t \text{ and } C_j > 0\} = i$, et donc $k = k'$.

Par ailleurs, comme σ_τ et $\sigma_{\tau'}$ se correspondent, alors il y a autant de x que de $\bar{x}$ entre les deux. En particulier, on ne peut pas avoir un $C_j > 0$ et $m_j = x$ pour $i < j \leq i'$, et si $m_i = x$ alors $C_i > 0$.

Réciproquement, supposons qu'il existe un t qui ne satisfait pas les conditions du lemme. Soit i l'indice du bloc courant à l'instant τ, et i' à l'instant t. Si $i = i'$ la suite d'extraction n'est pas croissante et on détecte une erreur. Sinon, alors $m_{i'} < x$ et donc nécessairement $k' \geq i' > i = k$. □

Proposition 9. *L'algorithme 6 est un algorithme randomisé dans le modèle de streaming en une passe de la droite vers la gauche pour* PrioritYQueue *en espace $\tilde{O}(r)$. Si l'entrée est une file de priorité bien formée alors l'algorithme accepte toujours, sinon il refuse avec probabilité au moins $1 - n^{-c}$.*

Démonstration. Supposons $\sigma \in$ PrioritYQueue. Alors d'après le lemme 8, toute insertion est dans le même h_k que l'extraction correspondante. Donc en particulier, à la fin tous les h_k valent 0, et l'algorithme accepte. On a la complétude.

Pour prouver la correction, on ignore comme pour les preuves précédente la probabilité qu'un h_j soit évalué à 0 si les valeurs ajoutés et soustraites ne sont pas les mêmes, quand en réalité elle est d'au plus n^{-c}. Quand on dit que l'algorithme rejette, il rejette donc en réalité avec probabilité au moins $1 - n^{-c}$.

Si $\sigma \notin$ MultiSetEquality, on rejette pour les mêmes raisons que d'habitude.

Si $\sigma \in$ MultiSetEquality mais qu'il existe une extraction $\sigma_\tau = \bar{x}$ ne correspondant à aucune insertion σ_t pour $t < \tau$, alors si i est l'indice du bloc où se situe cette extraction, h_i contiendra au moins un $\bar{x}$ de plus que de x. Donc en particulier $h_i \neq 0$ et on rejette.

Si chaque extraction a bien une insertion qui lui correspond et se situe à gauche, mais que σ contient $\sigma_\tau = \bar{x}$, $\sigma_t = y$, $\sigma_{\tau'} = x$ avec σ_τ et $\sigma_{\tau'}$ qui se correspondent, $\tau' < t < \tau$ et $y < x$, alors par le lemme 8, σ_τ et $\sigma_{\tau'}$ sont comptés dans h_k et h'_k avec $k < k'$, donc à la fin $h_k = 0$.

Comme pour l'algorithme 1, on maintient r hashcodes de taille logarithmique et r compteurs C_i, donc on a un algorithme en espace $\tilde{O}(r)$. □

5.2 Fusion des blocs

Pour économiser de la place en mémoire, plutôt que d'utiliser des blocs de taille constante comme dans l'algorithme 5 ou les blocs que l'entrée impose comme dans les algorithmes 1 et 6, on va utiliser des blocs que l'on fusionne petit à petit en blocs plus gros, ce qui garantit de ne pas avoir trop de hashcodes à retenir.

On procède ainsi : chaque nouvelle valeur lue est un bloc. Puis, si les deux derniers blocs ont la même taille, on les fusionne et on répète le processus jusqu'à ce que tous les blocs soient de taille différente, comme le montre la figure 6. Ainsi après avoir lu t valeurs on a autant de blocs que le nombre de 1 dans l'écriture binaire de t.

Pour garantir que les blocs sont les mêmes dans les deux sens, on rajoute des $0.\bar{0}$ à la fin de σ jusqu'à ce que $|\sigma|$ soit une puissance de 2. Ainsi on a exactement le même découpage à tous les niveaux.

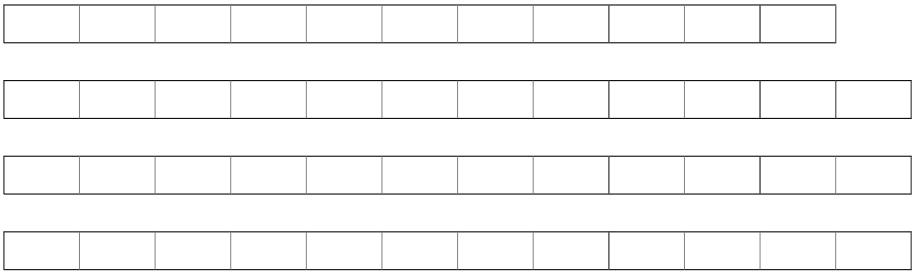

FIGURE 6 – Processus de fusion des blocs

Un bloc, comme dans les algorithmes précédents, correspond à des éléments consécutifs de σ. Cependant, comme précédemment, les éléments qui sont dans son hashcode ne sont pas forcément les éléments du bloc.

En mémoire, un bloc est caractérisé par :

- h le hashcode qui lui est associé
- m le minimum des extractions du bloc, qui sert comme pour les algorithmes précédents à décider dans quel hashcode les nouveaux éléments doivent être mis, et également à vérifier que les suites d'extractions sont décroissantes
- l la longueur du bloc, qui sert à décider quand les deux derniers blocs doivent être fusionnés
- δ la différence entre le nombre d'insertions et d'extractions qui sont comptées dans h, qui sert à décider quand on peut tester h à 0

En effet, comme à la fin de l'algorithme il ne reste qu'un seul bloc, et qu'un seul hashcode, on doit faire les tests à 0 à la volée. Par conséquent il arrive qu'un test à 0 survienne "trop tard", quand l'erreur a déjà été absorbée dans une fusion de bloc. Cependant l'esprit de l'algorithme est de garantir que dans au moins un des deux sens on détectera l'erreur lors de la fusion.

La figure 7 illustre un cas de deux files de priorités, la première fausse et la seconde bien formée, que la première passe de l'algorithme n'arrive pas à distinguer. En effet, au moment où l'algorithme arrive au $\overline{2}$ qui ne devrait pas être mis dans le bon hashcode, ce qui déclencherait un rejet, la mémoire ne contient plus qu'un seul hashcode non vide.

La figure 8 montre que dans la seconde passe, de droite à gauche, le 2 va être mis dans le dernier hashcode, qui ne contient que $\overline{3}$. Comme le δ de ce bloc va devenir 0, le hashcode va être testé à 0 et l'algorithme va rejeter.

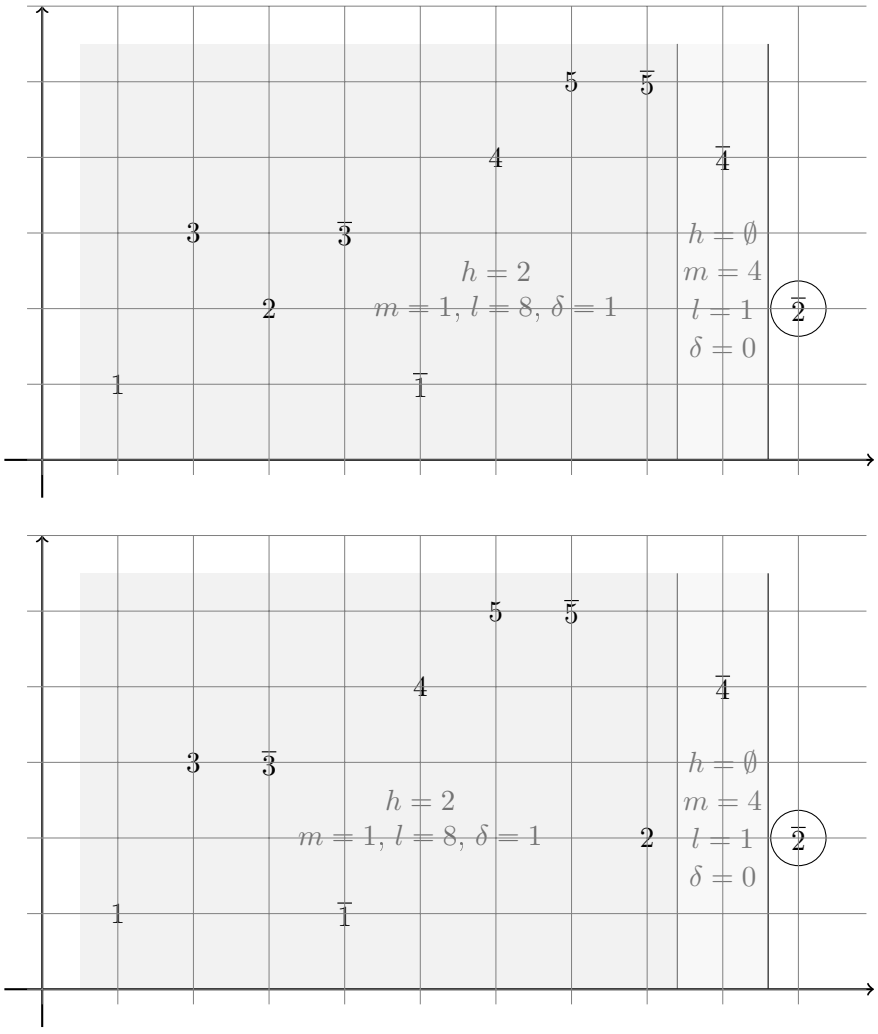

FIGURE 7 – Ici, lorsqu'on arrive à $\overline{2}$ l'information du premier bloc est oubliée et on ne sait pas faire la différence entre un passé correct et un passé faux.

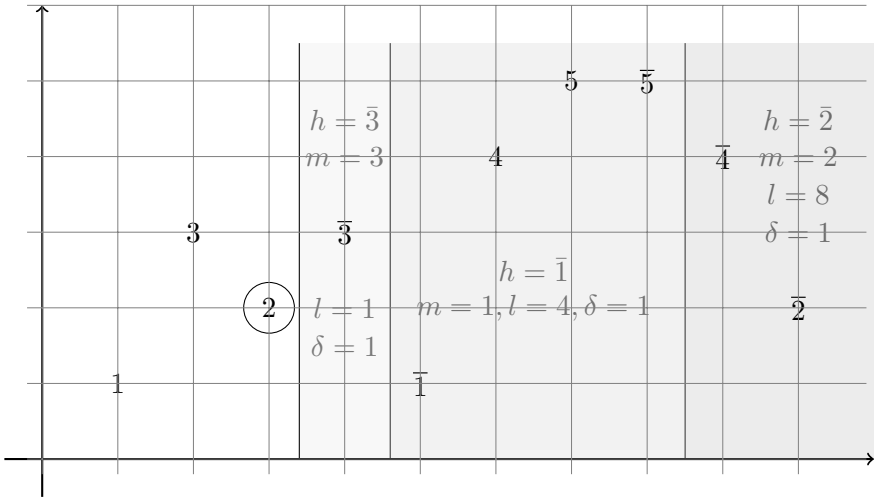

FIGURE 8 – Lors de la passe de la gauche vers la droite, on détecte l'erreur et on rejette.

5.3 Résultat

Théoreme 10. *L'algorithme 9 (en annexe A) est un algorithme randomisé dans le modèle de streaming en une passe pour* PRIORITYQUEUE *en espace* $O((\log n)^2)$. *Si l'entrée est une file de priorité bien formée alors l'algorithme accepte toujours, sinon il refuse avec probabilité au moins* $1 - n^{-c}$.

La preuve du théorème 10 a été effectuée lors du stage, mais elle est longue et fastidieuse et n'entre pas dans le cadre du rapport.

A Détail de l'algorithme à deux passes

On présente ici l'algorithme en deux passes en espace $O((\log n)^2)$ pour PriorityQueue. L'analyse de cette algorithme a été effectuée durant le stage mais n'est pas reproduite ici pour des raisons de place.

Algorithme 9 Algorithme en deux passes en espace $O((\log n)^2)$

$q \leftarrow \lceil \log n \rceil$; $\sigma \leftarrow \sigma(0\bar{0})^{\frac{2^q - n}{2}}$
Algorithme 10(σ)
Algorithme 11(σ)
Accept

La passe de la gauche vers la droite n'utilise pas les C_x des autres algorithmes. En effet, détecter les extraction qui ne correspondent pas à une insertion précédente se fait naturellement plus facilement dans la passe de la droite vers la gauche.

Algorithme 10 Passe de la gauche vers la droite

$S \leftarrow []$; $t \leftarrow 0$
α random integer in $\{1, \ldots, p-1\}$
while $t < |\sigma|$ **do**
  $t \leftarrow t + 1$
  if σ_t is insert **then**
    ReadInsertLeft(σ_t)
  else
    ReadExtractLeft(σ_t)
    if σ_{t-1} is extract **then**
      Check($\sigma_{t-1} \geq \sigma_t$)
    end if
  end if
  while 2 last elements of S have same block size **do**
    $(h_i, m_i, l_i, \delta_i) \leftarrow$**Pop**($S$)
    $(h_{i-1}, m_{i-1}, l_{i-1}, \delta_{i-1}) \leftarrow$**Pop**($S$)
    Push($(h_i + h_{i-1} \mod p, min(m_i, m_{i-1}), l_i + l_{i-1}, \delta_i + \delta_{i-1}$,$S$)
  end while
end while
Check($S = [(0, 0, n, 0)$)
Accept

La passe de la droite vers la gauche utilise un argument de plus dans les blocs, C. Il caractérise le nombre d'extractions de m qui n'ont pas encore trouvé d'insertion correspondante.

Algorithme 11 Passe de la droite vers la gauche

$S \leftarrow [] \; ; t \leftarrow n$
α random integer in $\{1, \ldots, p-1\}$
while $t > 0$ **do**
  if σ_t is extract **then**
    ReadExtractRight(σ_t)
  else
    ReadInsertRight(σ_t)
  end if
  while 2 last elements of S have same block size **do**
    $(h_i, m_i, l_i, \delta_i, C_i) \leftarrow$**Pop**$(S)$
    $(h_{i-1}, m_{i-1}, l_{i-1}, \delta_{i-1}, C_{i-1}) \leftarrow$**Pop**$(S)$
    if $m_i = m_{i-1}$ **then**
      $C \leftarrow C_i + C_{i-1}$
    else
      if $m_i < m_{i-1}$ **then**
        $C \leftarrow C_i$
      else
        $C \leftarrow C_{i-1}$
      end if
    end if
    Push$((h_i + h_{i-1} \mod p, min(m_i, m_{i-1}), l_i + l_{i-1}, \delta_i + \delta_{i-1}, C, S)$
    if $m_{i-1} > m_i$ **then**
      Check$(h_i = 0)$
    end if
  end while
  $t \leftarrow t - 1$
end while
Check$(S = [(0, 0, n, 0, 0))$
Accept

Algorithme 12 ReadInsertLeft

$i \leftarrow \text{Size}(S)$
$k \leftarrow \max\{j | m_j < \sigma_t, j \leq i\}$
$h_k \leftarrow \text{hash}_+(h_k, \alpha, \sigma_t)$
$\delta_k \leftarrow \delta_k + 1$
Push$((0, +\infty, 1, 0), S)$

Algorithme 13 ReadExtractLeft

$i \leftarrow \text{Size}(S)$
$k \leftarrow \max\{j | m_j < \sigma_t, j \leq i\}$
$h_k \leftarrow \text{hash}_-(h_k, \alpha, \sigma_t)$
$\delta_k \leftarrow \delta_k - 1$
Check$(\delta_j \geq 0)$
if $\delta_k = 0$ **then**
  Check$(h_j = 0)$
end if
Push$((0, \sigma_t, 1, 0)$,$S)$

Algorithme 14 ReadExtractRight

Push$((\text{hash}_+(0, \alpha, \sigma_t), \sigma_t, 1, 1, 1)$,$S)$

Algorithme 15 ReadInsertRight

$i \leftarrow \text{Size}(S)$
$k \leftarrow \max\{j \leq i | m_j < \sigma_t \text{ or } (m_j = \sigma_t \text{ and } C_j > 0)\}$
$h_k \leftarrow \text{hash}_-(h_k, \alpha, \sigma_t)$
$\delta_k \leftarrow \delta_k - 1$
if $m_k = \sigma_t$ **then**
  $C_k \leftarrow C_k - 1$
end if
Check$(\delta_j \geq 0)$
if $\delta_k = 0$ **then**
  Check$(h_j = 0)$
end if
Push$((0, +\infty, 1, 0, 0)$,$S)$

Références

[CCKM10] A. Chakrabarti, G. Cormode, R. Kondapally, and A. McGregor. Information cost tradeoffs for augmented index and streaming language recognition. In *Proceedings of 51st IEEE Annual Symposium on Foundations of Computer Science*, pages 387–396. IEEE, 2010.

[MMN10] F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. In *Proceedings of 42nd ACM Symposium on Theory of Computing*, pages 261–270, 2010.