Projet 2 : 2017 - 2018

Structures de données et algorithmes

Prof. P. Geurts

Perin Alexis
Muramutsa Jean

April 26, 2018

1) Analyse théorique

1) Cas du tri

MqUpdate(M, x)

```
mq.circular[mq.start] = x
p = 0
for i = 1 to mq.size
    if mq.sorted[i] ≥ x
        p = i
        break
if mq.sorted[mq.size] < x
    p = mq.size
if p < mq.start
    for i = mq.start downto p
        mq.sorted[i] = mq.sorted[i − 1] End
    if p > mq.start
        for i = mq.start to p
            mq.sorted[i] = mq.sorted[i + 1]
    mq.sorted[p] = x
    mq.start = (mq.start + 1)%mq.size
```

Dans le meilleur cas, la première valeur supérieure à x dans le vecteur *circular* est aussi la plus ancienne valeur du vecteur. La première boucle for s'arrête à l'endroit du vecteur où cette valeur se trouve et la valeur x vient s'y écraser. Nous obtenons une complexité $\Theta(n)$.

Dans le pire cas, la plus ancienne valeur se trouve en première position dans le tableau et x étant supérieur à toutes les valeurs du vecteur, devra se placer à la fin de celui-ci. La complexité dans le temps est $\Theta(n)$.

2) Variante du QuickSort

MqUpdate(M, x)

```
mq.circular[mq.start] = x
mq.start = (mq.start + 1)%mq.size
for i = 1 to mq.size
    mq.sorted[i] = mq.circular[(mq.start + i)%mq.size]
i = 0
while mq.size − i > (mq.size + 1)/2
    QuickSort(mq.sorted, 1, mq.size − i)
    i = i + 1
```

Le QuickSort employé ici n'effectue d'appel récursif que sur la partie du vecteur qui suit la nouvelle position du pivot (initialement la dernière valeur du tableau) obtenue après l'emploi de la fonction *Partition*. Pour un filtre de taille ω, la fonction QuickSort est appelée $(\omega-1)/2$ fois sur un sous-vecteur qui pert à chaque tour de boucle la dernière valeur du vecteur utilisé précédemment. Cela assure non seulement que les $(\omega+1)/2$ valeurs du sous-vecteur obtenu en fin de boucle sont les plus petites valeurs du filtre mais aussi que la dernière valeur de se sous-vecteur est la plus grande de celles-ci, donc la médiane du filtre.

Dans le meilleur comme le pire cas, la condition de la boucle *while* est testée $(\omega+1)/2$ fois et la fonction QuickSort est appelée une fois de moins et il en va de même pour le nombre de fois que l'indice i est incrémenté.

Dans le meilleur cas, à chaque tour de boucle, le pivot est déplacé au milieu du vecteur. Les différents tris successifs nous donnent une complexité dans le temps $\Theta(\omega^2 log\omega)$.

Dans le pire cas, le pivot est toujours déplacé après l'appel de *Partition* et devient la première valeur du tableau, le reste du tableau étant alors trié récursivement. Le problème est donc $\Theta(\omega^3)$.

3) Fonction HeapReplace

HEAPREPLACE($heap, toReplace, value$)

$newKey = heap.indexes[toReplace.key]$
$heap.arr[new_key].val = value$
downHeapify($heap, newKey$)
upHeapify($heap, newKey$)

Les fonctions $downHeapify$ et $upHeapify$ servent à créer un tas-min ou un tas-max avec en respectant les propriétés respectives des tas. Cela est fait, au besoin, avec des opérations de type *swap* (opérations $\mathcal{O}(1)$).

Dans le meilleur cas, celui où après le remplacement d'une valeur, les propriétés du tas sont toujours conservées, la fonction HeapReplace est $\Theta(1)$.

Dans le pire cas, x va se placer sur la dernière feuille de l'arbre, alors qu'en respectant les propriétés de l'arbre, elle devrait se trouver à la racine de celui-ci. Avec un filtre de taille ω, on peut créer un arbre de hauteur $\log_2 \omega$. On en conclut que le problème est $\Theta(\log_2 \omega)$.

4) Cas des tas

MQUPDATE(M, x)

if ($mq.circular[mq.start].type$)
    heapReplace($mq.maxHeap, mq.circular[mq.start].ref, x$)
else
    heapReplace($mq.minHeap, mq.circular[mq.start].ref, x$)
if heapTop($mq.minHeap$) < heapTop($mq.maxHeap$)
    swapTops(mq)
$mq.start = (mq.start + 1)\%mq.size$

La fonction $swapTops(mq)$ permet d'échanger les racines des tas-max et tas-min formés à partir des valeurs contenues dans la MedianQueue. Dans le meilleur cas, la nouvelle valeur x est ajoutée à la MedianQueue en remplaçant sa plus ancienne valeur et l'ordre des deux tas est toujours conservé, avec la récine du tas-min restant plus élevée que celle du tas-max. Dans ce cas, seule la fonction HeapReplace est utilisée et dans son meilleur cas. Ainsi, nous obtenons une complexité dans le temps $\Theta(1)$.

Dans le pire cas, nous avons la situation inverse : x est placé sur la dernière feuille de l'un des deux arbres. Après tri, l'on constate que la condition qui lie les sommets des deux arbres n'est pas respectée et x est échangé avec la racine de l'autre arbre. Les opérations effectuées dans *swapTops* étant $\Theta(n)$ et le pire cas de *heapReplace* étant $\Theta(\log_2 \omega)$, nous conclusons que le pire cas de $MqUpdate$ dans ce cas-ci est $\Theta(\log_2 \omega)$.

5) Fonction SigMedian

La fonction la plus coûteuse appelée dans $SignalMedian$ est $MqCreate$. Un meilleur et un pire cas n'étant pas diistingables, nous concluons que la fonction *SignalMedian* a une complexité moyenne dans le temps par rapport à ω de $\mathcal{O}(\omega^2)$, provenant de l'utilisation de *heapAdd* dans une boucle sur toute la longueur de la fenêtre. L'influence de la taille du signal est donnée par l'emploi de la fonction *mqUpdate* dans une boucle dont l'argument va s'incrémenter N - ω fois. Les complexités dans le meilleur et le pire cas sont alors respectivement $\mathcal{O}(N)$ et $\mathcal{O}(N \log_2 \omega)$.

2) Analyse empirique

La méthode des tas est la plus rapide pour des signaux de plus petites tailles et des filtres de taille prèsque égale en raison de sa complexité dans le temps, moins élevée dans le meilleur et le pire cas par rapport aux autrs méthodes, dans ces conditions. Si l'on augmente la taille du signal et qu'un écart se crée avec la taille du filtre, la première méthode (tri) s'avère être la plus efficace.