Université de Liège

Projet 2 - Filtre médian

Structure de données et algorithmes

Maxime Meurisse
Valentin Vermeylen

2e année de Bachelier Ingénieur Civil
Année académique 2017-2018

1 Analyse théorique

1.1 Variante basée sur le tri

Le pseudo-code de la fonction MQUPDATE dans le cas du tri est donné à la figure 1. La fonction SWAP utilisée permet d'échanger les positions de deux éléments d'un tableau.

```
MQUPDATE (mq, value)
 1  i = 1
 2  while mq.sorted[i] ≠ mq.circular[mq.start]
 3        i = i + 1 ;
 4  mq.sorted[i] = value
 5  mq.circular [mq.start] = value
 6  mq.start = (mq.start + 1) %mq.size
 7  if i > 1
 8        if mq.sorted[i − 1] > mq.sorted[i]
 9              for j = i downto 2
10                    if mq.sorted[j − 1] > mq.sorted[j]
11                          SWAP(mq.sorted[j − 1], mq.sorted[j])
12  if i < mq.size
13        if mq.sorted[i + 1] < mq.sorted[i]
14              for j = i to mq.size − 1
15                    if mq.sorted[j + 1] < mq.sorted[j]
16                          SWAP(mq.sorted[j + 1], mq.sorted[j])
```

Figure 1 – Pseudo-code de la fonction MQUPDATE dans le cas du tri.

1.1.1 Complexité en temps dans le pire cas

Dans le pire cas, la nouvelle valeur insérée dans le tableau `sorted` n'est pas à sa bonne position et doit être amenée à l'autre bout du tableau. Dans ce cas, celle-ci devra faire $w-1$ échanges de position. Toutes les opérations étant $O(1)$, la complexité en temps sera $O(w-1) = O(w)$.

1.1.2 Complexité en temps dans le meilleur cas

Dans le meilleur cas, la nouvelle valeur est insérée à la première position du tableau `sorted` et est déjà à sa bonne position. L'algorithme ne rentrera dans aucune boucle, la complexité en temps sera donc $O(1)$.

1.2 Variante basée sur le QuickSort

Dans cette variante, la fonction mqUpdate fait appel à la fonction adaptedQuicksort après avoir mis à jour les valeurs du tableau `sorted` de la structure `mq`.

Le pseudo-code de la fonction adaptedQuicksort est donné à la figure 3. La fonction swap est identique à celle présentée au point 1.1 et la fonction partition est celle utilisée pour la version classique du QuickSort, implémentée pour le projet 1.

```
mqUpdate(mq, value)
1  mq.circular [mq.start] = value
2  mq.start = (mq.start + 1) %mq.size
3  for i = 1 to mq.size
4        mq.sorted [i] = mq.circular [(mq.start + i) %mq.size]
5  adaptedQuickSort(mq, 1, mq.size)
```

Figure 2 – Pseudo-code de la fonction mqUpdate dans le cas de la variante du quicksort.

```
adaptedQuicksort (mq, first, last)
1  if first < last
2        posPivot = partition (mq.sorted, first, last)
3        posMedian = mq.size/2
4        if posPivot ≠ posMedian
5              if posPivot > posMedian
6                    adaptedQuicksort(mq, first, posPivot − 1)
7              else
8                    adaptedQuicksort(mq, posPivot + 1, last)
```

Figure 3 – Pseudo-code de la fonction adaptedQuicksort.

1.2.1 Complexité en temps dans le pire cas

Dans tous les cas, il faut charger les valeurs dans le tableau `sorted`. Cette étape est $O(w)$.

La fonction partition, comme vu au cours, est $\Theta(n)$ avec n le nombre d'éléments dans la tableau à partitionner.

Dans le pire cas, il va falloir appeler w fois adaptedQuicksort, les pivots étant alternativement les derniers et premiers éléments de chaque nouveau tableau, de sorte que le pivot ne devienne la médiane qu'au dernier élément.

La formule de la complexité de adaptedQuicksort étant identique, dans le pire cas, à celle de QuickSort dans le pire cas ($T(n) = T(n-1) + c$), on trouve que adaptedQuicksort est $O(w^2)$ dans le pire cas.

La fonction mqUpdate a donc une complexité en temps de $O(w^2)$ dans le pire cas.

1.2.2 Complexité en temps dans le meilleur cas

Dans le meilleur cas, on boucle quand même puis on ne rentre dans le adaptedQuicksort qu'une fois, on a donc une complexité de $O(w)$ due à la boucle.

1.3 La fonction heapReplace

Le pseudo-code de la fonction heapReplace est donné à la figure 4. Les fonctions maxHeapify et minHeapify sont identiques à celles vues au cours.

```
heapReplace(heap, toReplace, value)
1  pos = toReplace.index
2  heap.referenceArray[pos].value = value
3  for i = heap.nbElements/2 to i = 1
4       if heap.type
5            maxHeapify(heap.referenceArray, i, heap.nbElements)
6       else
7            minHeapify(heap.referenceArray, i, heap.nbElements)
```

Figure 4 – Pseudo-code de la fonction heapReplace.

1.3.1 Complexité en temps dans le pire cas

Dans le pire cas, l'élément placé se trouve en sommet de tas et doit se retrouver dans une feuille pour respecter la propriété d'ordre du tas (ou l'inverse selon le type du tas et les valeurs présentes).

Un tas de n éléments possède une hauteur de $\lfloor \log_2 n \rfloor$, qu'il soit complet ou non étant donné qu'il est représenté par un arbre binaire complet.

Ainsi, dans le pire cas, pour un noeud, on a $\log n$ appels récursifs à maxHeapify (ou minHeapify). Ces opérations font appel à la méthode swap. Dans le pire cas, celle-ci effectue une recherche linéaire dans tout le tableau pour trouver les positions des éléments à échanger, elle est donc $O(n)$. La méthode heapReplace appelant $O(n)$ fois les méthodes maxHeapify (ou minHeapify) récursivement, on a donc une complexité finale de $O(n^2 \log n)$ dans le pire cas.

Remarque Cette complexité élevée pourrait être améliorée en évitant d'appeler autant de fois les méthodes MAX-MINHEAPIFY. On pourrait en effet se contenter d'échanger la valeur du noeud avec son parent ou avec un enfant s'il en possède et que la propriété d'ordre du tas n'est pas vérifiée. Tout le reste du tableau respectant déjà la propriété d'ordre des tas, on n'aurait qu'à effectuer au plus $\log n$ swaps sans parcourir tout la tableau. La complexité dans le pire cas deviendrait ainsi $O(n \log n)$ et non plus $O(n^2 \log n)$ et dans le meilleur cas, on atteindrait une complexité de $O(1)$. Néanmoins, cette implémentation nous provoquait des erreurs dans le fichier de sortie et nous n'avions plus le temps de la débugger.

1.3.2 Complexité en temps dans le meilleur cas

Dans le meilleur cas, la valeur est insérée de telle manière que, si on a affaire à un tas-max, elle est plus grande ou égale au plus grand de ses fils et plus petite ou égale à son parent (si elle en a).

La fonction HEAPREPLACE est dans ce cas $O(n)$ car on ne doit pas l'échanger avec quelque élément que ce soit. La fonction ne fait que remplacer la valeur précédente et vérifier que la propriété d'ordre du tas est bien respectée (en appelant, dans une boucle, les fonctions MAX-MINHEAPIFY qui ne font aucune opération puisque la valeur est déjà bien placée dans le tas.), ce qui est le cas initialement dans le meilleur cas.

Le raisonnement est similaire dans le cas d'un tas-min.

La complexité au meilleur cas de HEAPREPLACE est donc $O(n)$.

1.4 Variante à base de tas

Le pseudo-code de la fonction MQUPDATE dans le cas des tas est donné à la figure 5.

1.4.1 Analyse des complexités

Presque toutes les opérations sont $O(1)$, à l'exception de HEAPREPLACE qui est $O(n^2 \log n)$ ou $O(n)$ selon le pire ou meilleur cas.

Dans le meilleur cas, on ne boucle qu'une fois, on ne doit rien échanger (on remplace par la même valeur au sommet du tas, par exemple) et on ne doit pas échanger avec l'autre tas. La complexité en temps est donc $O(n)$.

Dans le pire cas, on parcourt les deux tas, la valeur à remplacer est la dernière du deuxième tas, elle doit tout remonter et tout remonter dans le second tas. On reste néanmoins en $O(n^2 \log n)$.

Chaque tas contenant une moitié des éléments du filtre de taille w, dans chaque expression de la complexité, n peut être remplacé par $\frac{w}{2}$.

```
mqUpdate (mq, value)
 1  r = mq.circular[mq.start]
 2  done = false;
 3  for i = 0 to heapSize(mq.maxHeap)
 4      if heapGet(mq.maxHeap, referenceArrayIndex(mq.referenceMaxHeap, i)) == r
 5          heapReplace(mq.maxHeap, referenceArrayIndex(mq.referenceMaxHeap, i), value)
 6          done = true;
 7
 8  if (!done)
 9      for i = 0 to heapSize(mq.minHeap)
10          if heapGet(mq.minHeap, referenceArrayIndex(mq.referenceMinHeap, i)) == r
11              heapReplace(mq.minHeap, referenceArrayIndex(mq.referenceMinHeap, i), value)
12
13  if heapTop(mq.maxHeap) > heapTop(mq.minHeap)
14      k = 0, m = 0
15      topMax = heapTop(mq.maxHeap, topMin = heapTop(mq.minHeap
16      while heapGet(mq.maxHeap, referenceArrayIndex(mq.referenceMaxHeap, k)) ≠ topMax
17          k = k + 1;
18      while heapGet(mq.minHeap, referenceArrayIndex(mq.referenceMinHeap, m)) ≠ topMin
19          m = m + 1;
20      heapReplace(mq.maxHeap, referenceArrayIndex(mq.referenceMaxHeap, i), topMin)
21      heapReplace(mq.minHeap, referenceArrayIndex(mq.referenceMinHeap, i), topMax)
22  mq.circular [mq.start] = value
23  mq.start = (mq.start + 1) %mq.size
```

Figure 5 – Pseudo-code de la fonction mqUpdate dans le cas des tas.

1.5 La fonction sigMedian

Toutes les opérations sont $O(1)$, à l'exception de mqCreate qui est la complexité de création de `mq`, et de la recherche de la médiane ainsi que mqUpdate, qui sont calculées $N - w$ fois.

Dans le cas d'un N suffisamment élevé, c'est la complexité de mqUpdate qui est décisive pour le calcul de la complexité et non celle de mqCreate qui ne fait qu'assigner 2 tableaux. Les complexités en temps pour sigMedian vaudront donc $(N - w)*$ les complexités de mqUpdate dans les différentes implémentations et pour les différents cas.

Les complexités de mqUpdate ont été calculées ci-avant (et sont reprises dans le tableau 1), et les complexités de mqMedian sont $O(1)$ pour tous les cas.

Algorithme	Pire	Meilleure
`SortedMedianQueue`	$O(n)$	$O(1)$
`QuickMedianQueue`	$O(n^2)$	$O(1)$
`HeapReplace`	$O(n^2 \log n)$	$O(n)$
`HeapMedianQueue`	$O(n^2 \log n)$	$O(n)$

Tableau 1 – Résumé des complexités des différentes implémentations.

2 Analyse empirique

2.1 Comparaison des différentes implémentations

Afin de comparer empiriquement les différentes implémentations, les algorithmes ont été testés sur le signal `temperatures23.03.17_23.03.18.sgl` pour différentes tailles w de filtre.

Les temps reportés dans le tableau 2 sont des moyennes de 10 temps et s'expriment en secondes.

w	Naive	Sorted	Quick	Heap
5	0,001 134	0,000 391	0,001 116	0,001 972
25	0,009 255	0,001 142	0,006 579	0,005 979
105	0,054 978	0,003 854	0,029 643	0,021 637
505	0,325 518	0,015 914	0,136 352	0,086 637
1005	0,669 930	0,031 882	0,275 488	0,181 638
2005	1,242 557	0,055 246	0,555 251	0,360 648
5005	1,833 140	0,067 319	0,998 436	0,531 199

Tableau 2 – Comparaison des temps d'exécution, en secondes, des différentes implémentations sur un signal pour différentes tailles w de filtre.

À première vue, l'algorithme SortedMedianQueue semble le meilleur pour toutes les tailles de filtres. Les algorithmes QuickMedianQueue et HeapMedianQueue (bien que celui-ci pourrait être amélioré tel que précisé dans une remarque précédemment) se suivent d'assez près et semblent être efficace, bien qu'ils soient tout de même beaucoup plus lents que Sorted. L'algorithme NaiveMedianQueue est de loin le plus inefficace.

Pour de grandes tailles de signaux, l'implémentation basée sur le Quicksort fournit des temps qui augmentent de manière plus importante que dans le cas des autres implémentations. Cela peut être dû au fait que le QuickSort se comporte moins bien s'il doit trier des tableaux contenant peu de valeurs différentes et, sur 5000 valeurs, on retrouvera de nombreuses fois les mêmes températures.

2.2 Comparaison avec les résultats théoriques

Au vu des résultats, on voit que la variante basée sur le quicksort augmente presque linéairement, ce qui doit correspondre au cas moyen ($n \log n$). La méthode naïve semble augmenter de manière un peu plus que linéaire et prend initialement un temps plus élevée que les autres pour fournir ses valeurs car la totalité du tableau est rechargé à chaque itération. Néanmoins, elle utilise la fonction de tri qsort qui est souvent implémenté grâce à un quicksort, donc de $n \log n$, ce qui est légèrement supérieur à n (on a ici remplacé w par n).

La méthode basée sur l'utilisation de tas semble aller à l'encontre de sa complexité théorique en exhibant un comportement presque linéaire, ce qui se rapprocherait du meilleur cas. Cela peut se comprendre car le pire cas est fort spécifique et a bien moins de probabilité de se produire qu'un cas moyen, qui serait proche du meilleur cas (échange d'un seul ou de deux noeuds par exemple).

Finalement, la méthode basée sur le tri fournit les meilleurs résultats, ce qui est parfaitement compréhensible au vu de sa complexité théorique ($O(n)$ dans le pire cas et $O(1)$ dans le meilleur).

Remarque Il serait possible d'implémenter la méthode basée sur les tas de manière à obtenir une complexité de $\log n$ mais nous n'y sommes pas parvenu tout en conservant l'opacité des structures (les recherches linéaires rendent cela impossible).