Median Filtering is Equivalent to Sorting

Jukka Suomela · Aalto University
TU Berlin · 6 August 2014
Median filter

input: \(n\) elements
window size: \(k\)
output: \(n-k+1\) medians

a.k.a. sliding window median, moving median, running median, rolling median, median smoothing
Median filter

• In numerous scientific computing systems:
  • \textit{R}: “runmed”
  • \textit{Mathematica}: “MedianFilter”
  • \textit{Matlab}: “medfilt1”
  • \textit{Octave}: “medfilt1” (\texttt{signal} package)
  • \textit{SciPy}: “medfilt1” (\texttt{scipy.signal} module)
Median filter

- In numerous scientific computing systems:

- 2D version in image processing:
  - \textit{Photoshop}: “Median” filter
  - \textit{Gimp}: “Despeckle” filter
Prior work

- **Trivial:**
  - compute each median separately
  - $O(nk)$

- **“Streaming approach”:**
  - maintain a sliding window
  - $O(n \log k)$
Prior work

- “Streaming approach”
- Sliding window data structure, supports operations:
  - “find median”
  - “remove oldest and add new element”
Prior work

- Sliding window data structures for $B$-bit integers:
  - histogram with $2^B$ buckets
  - with linear scanning: $O(n2^B)$
  - with binary trees: $O(nB)$
  - with van Emde Boas trees: $O(n \log B)$

$n$: input size
$k$: window size
Prior work

- General sliding window data structures:
  - maxheap-minheap pair: $O(n \log k)$
  - binary search trees: $O(n \log k)$
  - finger trees: $O(n \log k)$
  - doubly-linked lists: $O(nk)$
  - sorted arrays: $O(nk)$

$n$: input size
$k$: window size
Prior work

• Maxheap-minheap pair
  • Astola–Campbell (1989)
  • Juhola et al. (1991)
  • Härdle–Steiger (1995) …

• Fast in practice

• Fast in theory, $O(n \log k)$ comparisons

$n$: input size
$k$: window size
Lower bounds

• For comparison-based algorithms: \(O(n \log k)\) is optimal
  • Juhola et al. (1991)
    Krizanc et al. (2005) …

• Reduction from sorting

\(n\): input size
\(k\): window size
State of the art

- $O(n \log k)$ comparisons is optimal
  - known since 1990s
  - nothing more to do here, case closed, problem solved

$n$: input size
$k$: window size
State of the art

- $O(n \log k)$ comparisons is optimal
- But we also know that $O(n \log n)$ comparisons is optimal for sorting in the worst case, yet this is not the full story!
  - integer sorting, adaptive sorting, cache-efficient sorting, GPU sorting …
State of the art

And what about implementations…

- $\mathcal{R}$: $\approx O(n \log k)$
- Mathematica: $\approx O(nk)$
- Matlab: $\approx O(nk)$
- Octave: $\approx O(nk)$
- SciPy: $\approx O(nk)$

why?! didn’t we do better already in 1980s?

$n$: input size
$k$: window size
Key idea

- Prior work:
  - “median filtering is as hard as sorting”

- Could we prove a matching upper bound:
  - “median filtering is as easy as sorting”
Key idea

• If we could show that:
  • “median filtering is equivalent to sorting”

• Then we could apply everything that we know about sorting here!
  • adaptive sorting $\rightarrow$ adaptive median filter
  • integer sorting $\rightarrow$ integer median filter …
Key idea

• If we could show that:
  • “median filtering is equivalent to sorting”

• Then we could apply everything that we know about sorting here!
  • all scientific computing packages know how to sort efficiently
Sorting-based lower bound

- Piecewise sorting: sort $n/k$ blocks of size $k$
  - with comparison sort: $O(n \log k)$ optimal
Sorting-based lower bound

- - 7 2 5 + + - - 1 4 3 + + - - 9 6 8 + +

2 5 7 5 5 1 1 1 3 4 4 3 9 6 6 8 9

2 5 7 1 3 4 6 8 9

pad with ±∞

median filter
Sorting-based lower bound

- Piecewise sorting: sort $n/k$ blocks of size $k$
  - with comparison sort: $O(n \log k)$ optimal
- Can be solved with $O(1)$ median filter operations
  - and some preprocessing & postprocessing

$n$: input size

$k$: window size
Sorting-based median filter

- Piecewise sorting: sort \( \frac{n}{k} \) blocks of size \( k \)
- Prior work:
  - median filter \( \approx \) as hard as piecewise sorting
- This work:
  - median filter \( \approx \) as easy as piecewise sorting

\( n \): input size
\( k \): window size
Sorting-based median filter

• High-level idea:
  • preprocessing = piecewise sorting
  • median filtering now possible in linear time!

• Simple and efficient
  • works very well also in practice
Sorting-based median filter

- Prior work:
  - median filtering $\approx$ data structure problem
  - how to maintain sliding window efficiently?

- This work:
  - median filtering $\approx$ algorithm problem
  - how to preprocess data?
Sorting-based median filter

- How does piecewise sorting help? We only know one median per block...

Input: 9 2 4 1 6 5 0 3 8 7

Sorted blocks: 1 2 4 6 9 0 3 5 7 8

Output: 4 ?? ?? ?? ?? ?? 5
Sorting-based median filter

- Basic idea: maintain sorted doubly-linked lists for each block
Sorting-based median filter

- **Sliding window** = two sorted linked lists
Sorting-based median filter

- **Sliding window** = two sorted linked lists
Sorting-based median filter

- **Sliding window** = two sorted linked lists
Sorting-based median filter

- **Sliding window** = two sorted linked lists
Sorting-based median filter

- **Sliding window** = two sorted linked lists
Sorting-based median filter

- **Sliding window** = two sorted linked lists
Sorting-based median filter

- Maintain "median pointers" for each list (one of these is the median)
Sorting-based median filter

- Maintain "median pointers" for each list (one of these is the median)
Sorting-based median filter

- Maintain "median pointers" for each list (one of these is the median)
Sorting-based median filter

- Maintain “median pointers” for each list (one of these is the median)
Sorting-based median filter

- Maintain "median pointers" for each list (one of these is the median)
Sorting-based median filter

• Maintain “median pointers” for each list (one of these is the median)
Sorting-based median filter

• Median pointers:
  • straightforward in $O(1)$ time per element
  • cf. merge sort

• Sorted linked lists:
  • how to insert & delete in $O(1)$ time?
Sorting-based median filter

- *Deletions* are easy if we know what to delete: start with a sorted list + pointers to it.
Sorting-based median filter

- *Deletions* are easy if we know what to delete: start with a sorted list + pointers to it.
Sorting-based median filter

- *Deletions* are easy if we know what to delete: start with a sorted list + pointers to it
Sorting-based median filter

- *Deletions* are easy if we know what to delete: start with a sorted list + pointers to it
Sorting-based median filter

- **Deletions** are easy if we know what to delete: start with a sorted list + pointers to it
Sorting-based median filter

- *Deletions* are easy if we know what to delete: start with a sorted list + pointers to it
Sorting-based median filter

- **Asymmetry:**
  - deletions from sorted linked lists easy
  - insertions to sorted linked lists hard

- **Reverse time!**
  - insertions become deletions, easy
Sorting-based median filter

- Reverse time: insertions become deletions, easy to do if we start with a sorted list

\[\begin{array}{cccccc}
9 & 2 & 4 & 1 & 6 \\
1 & 2 & 4 & 6 & 9 \\
\end{array}\]

\[\begin{array}{cccccc}
5 & 0 & 3 & 8 & 7 \\
0 & 3 & 5 & 7 & 8 \\
\end{array}\]
Sorting-based median filter

- Reverse time: insertions become deletions, easy to do if we start with a sorted list
Sorting-based median filter

- Reverse time: insertions become deletions, easy to do if we start with a sorted list
Sorting-based median filter

• Reverse time: insertions become deletions, easy to do if we start with a sorted list

[Diagram showing a sequence of numbers with arrows indicating the flow of elements, with a highlighted section of numbers 4, 1, 6, 5, 0, 3, 8, 7, 5, 3, 0, 8, 7, 8.]
Sorting-based median filter

- Reverse time: insertions become deletions, easy to do if we start with a sorted list
Sorting-based median filter

- Reverse time: insertions become deletions, easy to do if we start with a sorted list

```
9 2 4 1 6
```

```
1 2 4 6 9
```
Sorting-based median filter

- Reverse time

- How does this help?
  - insertions become deletions, nice
  - deletions become insertions, bad

- Solution: reverse time again
Sorting-based median filter

• Reverse time again:
  insert = *undo deletion*
Sorting-based median filter

- Reverse time again: insert = *undo deletion*
Sorting-based median filter

- Reverse time again:
  insert = *undo deletion*
Sorting-based median filter

- Reverse time again: insert = *undo deletion*
Sorting-based median filter

- Reverse time again: insert = *undo deletion*
Sorting-based median filter

- Reverse time again:
  insert = *undo deletion*
Sorting-based median filter

- Shrinking list: start with a sorted list
  - process one element = one deletion
- Growing list: start with a sorted list
  - first delete each element in reverse order
  - process one element = undo one deletion
Undo deletions from doubly-linked lists

- Knuth (2000): “dancing links”

- Delete: \( \text{prev}[\text{next}[i]] \leftarrow \text{prev}[i] \)
  \( \text{next}[\text{prev}[i]] \leftarrow \text{next}[i] \)

- Undo: \( \text{prev}[\text{next}[i]] \leftarrow i \)
  \( \text{next}[\text{prev}[i]] \leftarrow i \)
Sorting-based median filter

- Preprocessing: piecewise sorting
- Sliding window = sorted doubly-linked lists
  - shrinking list: easy
  - growing list: reverse time twice
  - insert = undo deletion, easy with dancing links
Sorting-based median filter

- Optimal algorithm for any input distribution, for almost any model of computing
  - just use optimal sorting algorithm for this setting
  - then $O(n)$ time postprocessing suffices

- Matching lower bound
Sorting-based median filter

• Easy to implement
• Very fast
def create_array(n):
    return [None] * n

def sort_block(alpha):
    pairs = [(alpha[i], i) for i in range(len(alpha))]
    return [i for v, i in sorted(pairs)]

class Block:
    def __init__(self, h, alpha):
        self.k = len(alpha)
        self.alpha = alpha
        self.pi = sort_block(alpha)
        self.prev = create_array(self.k + 1)
        self.next = create_array(self.k + 1)
        self.tail = self.k
        self.init_links()
        self.m = self.pi[h]
        self.s = h

    def init_links(self):
        p = self.tail
        for i in range(self.k):
            q = self.pi[i]
            self.next[p] = q
            self.prev[q] = p
            p = q
        self.next[p] = self.tail
        self.prev[self.tail] = p

    def unwind(self):
        for i in range(self.k - 1, -1, -1):
            self.next[self.prev[i]] = self.next[i]
            self.prev[self.next[i]] = self.prev[i]
        self.m = self.tail
        self.s = 0

    def delete(self, i):
        self.next[self.prev[i]] = self.next[i]
        self.prev[self.next[i]] = self.prev[i]
        if self.is_small(i):
            self.s -= 1
        else:
            if self.m == i:
                self.m = self.next[self.m]
            if self.s > 0:
                self.m = self.prev[self.m]
                self.s -= 1

    def undelete(self, i):
        self.next[self.prev[i]] = i
        self.prev[self.next[i]] = i
        if self.is_small(i):
            self.m = self.prev[self.m]

    def advance(self):
        self.m = self.next[self.m]
        self.s += 1

    def at_end(self):
        return self.m == self.tail

    def peek(self):
        return float('Inf') if self.at_end() else self.alpha[self.m]

    def get_pair(self, i):
        return (self.alpha[i], i)

    def is_small(self, i):
        return self.at_end() or self.get_pair(i) < self.get_pair(self.m)

    def sort_median(h, b, x):
        k = 2 * h + 1
        B = Block(h, x[0:k])
        y = []
        y.append(B.peek())
        for j in range(1, b):
            A = B
            B = Block(h, x[j*k:(j+1)*k])
            B.unwind()
            for i in range(k):
                A.delete(i)
                B.undelete(i)
            if A.s + B.s < h:
                if A.peek() <= B.peek():
                    A.advance()
                else:
                    B.advance()
                y.append(min(A.peek(), B.peek()))
        return y

complete Python implementation
For $bh = 10^5$, the plot shows the time (seconds) for different half-window sizes $h$. The libraries and algorithms compared include Mathematica, SciPy, Matlab, R, Stuetzle, Octave, and MoveMedian. The y-axis represents time in seconds, ranging from $10^{-2}$ to $10^3$, and the x-axis represents the half-window size $h$, ranging from $10^0$ to $10^4$.
$bh = 10^8$, all generators

- HeapMedian
- SortMedian
Conclusions

• Median filtering $\approx$ piecewise sorting

• In theory and in practice

• arXiv:1406.1717