Median Filtering is Equivalent to Sorting

Jukka Suomela · Aalto University
Helsinki Algorithms Seminar · 23 October 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:
  - **R**: “runmed”
  - **Mathematica**: “MedianFilter”
  - **Matlab**: “medfilt1”
  - **Octave**: “medfilt1” (signal package)
  - **SciPy**: “medfilt1” (scipy.signal module)
Median filter

• In numerous scientific computing systems:
  • \textit{R, Mathematica, Matlab, Octave, SciPy} …

• 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
  • $O(n \log k)$

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

• “Streaming approach”

• Sliding window data structure, supports operations:
  • “find median”
  • “remove oldest and add new element”

$n$: input size
$k$: window size
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
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…
  - \( R: \approx O(n \log k) \)
  - \( \text{Mathematica}: \approx O(nk) \)
  - \( \text{Matlab}: \approx O(nk) \)
  - \( \text{Octave}: \approx O(nk) \)
  - \( \text{SciPy}: \approx O(nk) \)

\( n: \) input size
\( k: \) window size

why?!

\( \text{didn’t we do better already in 1980s?} \)
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 \( \frac{n}{k} \) blocks of size \( k \)
  - with comparison sort: \( O(n \log k) \) optimal
Sorting-based lower bound

median filter

pad with $\pm \infty$
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 $n/k$ blocks of size $k$

• Prior work:
  • median filter $\approx$ as hard as piecewise sorting

• This work:
  • median filter $\approx$ as easy as piecewise sorting
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...

\[
\begin{array}{cccccccc}
9 & 2 & 4 & 1 & 6 & 5 & 0 & 3 & 8 & 7 \\
1 & 2 & 4 & 6 & 9 & 0 & 3 & 5 & 7 & 8 \\
\end{array}
\]

input 

sorted blocks 

output
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

```
9 2 4 1 6
```

```
0 3 5 7 8
```
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
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
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

• 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
$bh = 10^5$

- Mathematica
- SciPy
- Matlab
- R, Stuetzle
- R, Turlach
- Octave
- HeapMedian
- TreeMedian
- SortMedian
- MoveMedian
$bh = 10^8$, all generators

- HeapMedian
- SortMedian
Conclusions

• Median filtering $\approx$ *piecewise sorting*

• In theory and in practice

• arXiv:1406.1717