Sliding Window Minimum Implementations

Sliding window minimum is an interesting algorithm, so I thought I would implement it in a bunch of different languages. This repository contains (or will contain) implementations of the algorithm in different programming languages. What follows is an explanation of the problem and the algorithm.

The repository is stored at <https://bitbucket.org/keegan_csmith/sliding-window-minimum>

A copy of this article is stored at<http://people.cs.uct.ac.za/~ksmith/articles/sliding_window_minimum.html>

Problem Introduction

The algorithm is sometimes also referred to as the Ascending Minima algorithm. I learnt the algorithm from a South African Computer Olympiad camp some years ago. I couldn't find any references to it in any journal, however there are some explanations on the Internet.

Given an array ARR and an integer K, the problem boils down to computing for each index i: min(ARR[i], ARR[i-1], ..., ARR[i-K+1]). The algorithm for this "slides" a "window" of size K over ARR computing at each step the current minimum. In other words the algorithm roughly follows this psuedocode:

for (i = 0; i < ARR.size(); i++) {

remove ARR[i-K] from the sliding window

insert ARR[i] into the sliding window

output the smallest value in the window

}

The sliding window algorithm does the remove, insert and output step in amortized constant time. Or rather the time it takes to run the algorithm is O(ARR.size()).

Naive Algorithms

Before I explain the O(1) solution for sliding window minimum, let me explain some alternative solutions which are suboptimal. The most straight-forward solution is for each index i in ARR, simply loop over the values from ARR[i-K+1] to ARR[i] and keep track of the minimum:

void brute\_force\_time(std::vector<int> & ARR, int K) {

for (int i = 0; i < ARR.size(); i++) {

int min\_value = ARR[i];

for (int j = i - 1; j >= min(i - K + 1, 0); j--)

min\_value = min(min\_value, ARR[j]);

std::cout << min\_value << ' ';

}

}

The above C++ code runs in O(NK) where N = ARR.size(). We can improve on this by using a sorted set to speed up the minimum value queries to logarithmic time:

void logarithmic\_time(std::vector<int> & ARR, int K) {

std::multiset<int> window;

for (int i = 0; i < ARR.size(); i++) {

window.insert(ARR[i]);

if (i - K >= 0)

window.erase(window.find(ARR[i - K]));

std::cout << \*window.begin() << ' ';

}

}

The window can have at most K elements, so the multiset insert, find and begin operations all take time O(logK). This gives us an overall time of O(NlogK).

We can improve the run-time by a constant factor by using a heap instead. All the heap operations (except finding the minima) have the same run-time complexity as the multiset versions, however heaps empirically perform better. We need to be able to remove arbitrary items in the heap, but the implementation of heaps in C++ (std::priority\_queue) does not support this operation. Luckily a technique I have mastered in real life helps us get around this: we delete lazily. For each element in the priority queue we also store what index it was inserted from. Then when we query what the smallest element is, we discard it if its index falls out of range of our current window.:

void logarithmic\_time2(std::vector<int> & ARR, int K) {

// pair<int, int> represents the pair (-ARR[i], i). We use -ARR[i] since

// priority\_queue is by default a max-heap, but we want a min-heap. By

// negating the value on insertion and removal we get a min-heap. One can

// also use the rather ugly

// std::priority\_queue< std::pair<int, int>,

// std::vector< std::pair<int, int> >,

// std::greater< std::pair<int, int> > >

std::priority\_queue< std::pair<int, int> > window;

for (int i = 0; i < ARR.size(); i++) {

window.push(std::make\_pair(-ARR[i], i));

while(window.top().second <= i - K)

window.pop();

std::cout << (-window.top().first) << ' ';

}

}

Note that deleting lazily can actually make this algorithm perform rather badly. If the input is N values from N to 1, then a value is never deleted from the priority queue leading to O(logN) insertions. However, on average this is empirically faster than the multiset version.

Sliding Window Minimum Algorithm

The idea of lazily deleting elements is a salient one, but by putting in a bit more effort when inserting an element into the window we can get amortized O(1) run-time. Say our window contains the elements {1, 6, 7, 2, 4, 2}. We want to add the element 5 to our window. Notice that all elements in the window greater than 5 will now never be the minimum in the window for future i values, so we might as well get rid of them. The trick to this is to store the numbers in a deque [[1]](http://people.cs.uct.ac.za/~ksmith/articles/sliding_window_minimum.html#id2) and whenever inserting a number x we remove all numbers at the back of the deque which are greater than equal to x. Notice that if the deque was sorted before inserting, it will still be sorted after inserting x. Since the deque starts off sorted, it remains sorted throughout the sliding window algorithm. So the front of the deque will always be the smallest value.

The front of the queue might have a value which shouldn't be in the window anymore, but we can use the lazy delete idea with the deques as well. Now each element of ARR is inserted into the deque and deleted from the deque at most once. This gives as a total run-time of O(N) for the algorithm (amortized O(1) per insertion/deletion). Pretty sweet:

void sliding\_window\_minimum(std::vector<int> & ARR, int K) {

// pair<int, int> represents the pair (ARR[i], i)

std::deque< std::pair<int, int> > window;

for (int i = 0; i < ARR.size(); i++) {

while (!window.empty() && window.back().first >= ARR[i])

window.pop\_back();

window.push\_back(std::make\_pair(ARR[i], i));

while(window.front().second <= i - K)

window.pop\_front();

std::cout << (window.front().first) << ' ';

}

}

Extensions

You can modify the algorithm by flipping >= to <= to get the sliding window maximum algorithm.

In fact this algorithm works on any totally ordered set. So the elements can be floats, sets, strings, etc. Essentially anything which has a <= operator that behaves "nicely".

Think you fully understand this algorithm, try solving these problems:

* Task "sound" at <http://www.boi2007.de/en/tasks>
* Task "pyramid" at <http://olympiads.win.tue.nl/ioi/ioi2006/contest/day1/> (medium to hard problem)

|  |  |
| --- | --- |
| [[1]](http://people.cs.uct.ac.za/~ksmith/articles/sliding_window_minimum.html#id1) | Double-Ended Queue. Supports constant time insertion, removal and lookups at the front and the back of the queue. |