Design and Analysis of Algorithms

Theory Project

Fractional Cascading

An algorithmic approach

By

IMT2019051 Mani Nandadeep Medicharla

IMT2019063 R Prasannavenkatesh

IMT2019525 Vijay Jaisankar

Contents

1 Abstract

In this paper, we investigate the *Fractional Cascading* technique in building range trees and for fast searching of an element in multiple arrays. In this venture, we introduce and examine *Linear Search*, *Binary Search*, *Bridge Building* and *Auxiliary lists*. We also look at some of the *applications* of this technique, and suggest data structures for its efficient realisation.

2 Problem Statement

Fractional cascading: You are given an input of k ordered lists of numbers, each of size n as well as a query value x. The problem's output is to return, for each list, True if the query value appears in the list and False if it does not. For example, if the input is:

(a) List L_1: $[3, 4, 6]$

(b) List L_2: $[2, 6, 7]$

(c) List L_3: $[2, 4, 9]$

and the query value is 4, then the expected output is $[True, False, True]$.
Give an algorithm to solve the fractional cascading problem.

3 Brute Force

3.1 Linear Search

Linear search is the most basic search technique, wherein we sequentially compare each array element to the target element. In the worst case of the target element not coinciding with *any* list element, the algorithm would reach the end of the list and we would report an unsuccessful search. As each element is compared at most once, the time complexity is $O(n)$, where n is the size of the list.

This algorithm forms the basis of the simplest solution to our problem: We just run linear search on each of the k lists. If we have q queries, this takes $O(q \cdot k \cdot n)$ time, which is a lot and in real world situations, if k, q, and n are even moderately large, the time taken would become

astronomical.

```
Data: k arrays of size n and a query element x
Result: Boolean array regarding whether element is present in the
        indiced array or not
Function LinearSearch(Array,x):
    for i ← 0 to n do
        if Array[i] == x then
            return True;
        end
    end
return False
Function Main:
    output = [] ;
    for i ← 0 to k do
        output.append(linearSearch(input[i],x);
    end
return 0
```

Note that, however, this approach does not take into account any relevant information given to us in the question which can speed up this algorithm. It is given that the *lists are sorted*, so we can exploit this property and employ a faster searching technique to solve this problem in a better way: *Binary Search*

4 Improved Brute Force

4.1 Binary Search

Binary search is another searching algorithm that works correctly only on sorted arrays.
It begins by comparing the target element with the element at the middle of the list.

- If they are equal, we have found the target in the list
- If the target is larger, and as the list is sorted, we must now turn our attention to the *right half* of the list

- Similarly, if the target is smaller, we must focus on the *left half* of the list.

In the worst case, Binary Search will take $O(\log n)$ comparisons, where n is the size of the list.

To improve the performance of the Brute Force subroutine, we replace the Linear Search subroutine with the aforementioned Binary Search subroutine. If we have q queries, this takes $O(q \cdot k \cdot \log n)$ time, which is certainly a lot better than the initial brute force algorithm, but can yet be improved further.

```
Data: k arrays of size n and a query element x
Result: Boolean array regarding whether element is present in the
        indiced array or not
Function BinarySearch(Array,x,left,right):
    if right ≥ left then
        mid=(left+right)/2;
        if Array[mid] == x then
            return True;
        end
        if Array[mid] > x then
            BinarySearch(Array,x,left,mid-1);
        end
        else
            BinarySearch(Array,x,mid+1,right);
        end
    end
    else
        return False;
    end
end Function
Function Main:
    output = [] ;
    for i ← 0 to k do
        output.append(BinarySearch(input[i],x);
    end
return 0
```

5 Bridge Building

5.1 Introduction to bridges

A *bridge* is a pointer from an element a_i of A_i to an element a_j of A_{i+1} where $|a_i - a_j|$ is **small**, where A is the list and a represents an element of the array. By *small*, we mean an element that is either of the same value, or with the smallest difference to the one considered as reference.

Once we locate the answer to a query in one array, we should be able to **follow a bridge** to a key that is close to the answer in the next array.
In the best case, we follow a bridge from the answer to the query in A_i to a key in A_{i+1} and then from there locate the answer in A_{i+1}, all in constant time. If we can do *this*,then once we have the answer in A_1 we can find the answer in the remaining $k-1$ sorted arrays in $O(k)$ time complexity.

From a technical standpoint, we *implement* this method as follows:
For every element e in the first array, give e a pointer to the element with the *same value* in the second array or if the value doesn't exist, the *predecessor*. This is called bridge building between A_i and A_{i+1}. Then, once we've found the item in the first array, we can just follow these pointers down in order to figure out where the item is in all the other arrays. To find the answer in A_1, we can just use a balanced binary search tree,thus making the overall time complexity of our algorithm $O(\log n + k)$ per query.

5.2 Algorithm

Data: k arrays of size n and a query element x

Result: Boolean array regarding whether element is present in the indiced array or not

Function `BuildBridges(`*Array,x*`):`

- **for** $i \leftarrow 0$ **to** $k-1$ **do**
  - **for** $j \leftarrow 0$ **to** n **do**
    - Build bridge from Array[i][j] to Array[i+1][x] where |Array[i+1][y] - Array[i][j]| is small. In this approach, if both predecessor and successor exists, then we take predecessor first.
  - **end**
- **end**

end Function

Function `Main`**:**

- output = [] ;
- BuildBridges(input,x) ;
- output.append(BinarySearch(Array[i],x)) ;
- Once the element is found in the first array, follow the bridge path till the final array and append it to the output. ;

return 0

5.3 Shortcomings

This method seems like a very interesting and efficient alternative to solve this problem. However, there are some glaring weaknesses to this approach, the most important one being the fact that certain classes of inputs render this method useless.

In particular, if a later list is completely *in between* two elements of the first list, we have to redo the entire search, as the pointer pre-processing gives us no information that we didn't already know.

Let's consider a simple example to elucidate this statement

Consider the case where $k = 2$. Everything would be better if only we could guarantee that the first list contained the right elements to give you useful information about the second array. We

could just merge the arrays, but if we did this in the general case we'd end up with a totally merged array of size kn, which is not so good if k is large.

Even if each time we find an answer in the sorted Array A_i, we follow the bridge pointer from it as well as from the key above it we are still left with the entire contents of catalog A_{i+1} to search. This continues through the entire set of k catalogs.If we search each catalog by doing a linear scan from the point at which the bridge told us to begin,the total search time will be O(nk).Even if we build a balanced search tree over all elements in $Ai + 1$ that appear between two consecutive bridge pointers from A_i the query time will still be O(klogn) which is similar to performing k binary searches to get the query element.

6 Fractional Cascading

6.1 Intuition behind Fractional Cascading

Because of the shortcomings of the *Bridge Building* algorithm, i.e the case where every element of the "below array is in between all the elements of the "above" array.(**Note**: We say that array 1 is "above" array 2 and array 2 is "below" array 1. Therefore, an Array j is below Array i if $j > i$. In this case, we either have to do k binary searches, or we have to merge all n elements of the below array recursively and maintain bridges, to get the output. This will warrant a sub-optimal, extra time complexity of $O(k \cdot n)$, as we are going through *each* array element iteratively to merge it.

To avoid this problem of merging all elements and getting $O(k \cdot n)$ time complexity, we start with the *lowest* list in the sequence and select every ith element, where i = $\frac{1}{\alpha}$ and insert it into the array above it while still maintaining sorted order.
We then mark that element as *promoted* and keep a pointer from it, to its original position in the bottom list. This operation of taking every ith element and *promoting* it to the array above it is called *cascading*, and since we are only promoting a fraction of the elements, the algorithm is called **fractional cascading**.

For selecting, α, we can choose a variety of fractions, however $\alpha = \frac{1}{2}$ seems most appropriate because, we will have to compare only 2 elements while searching, after prepocessing. This will also ensure that our time complexity stays low, and easily computable.

6.2 Algorithmic Approach to Fractional Cascading

Let the input be specified by k n-element arrays, $A_1, A_2, \ldots, A_k$, Let the query element value be x. Let $M_1, M_2, \ldots, M_k$ be the new *merged arrays* such that $M_k = A_k$ and $\forall i < k$, M_i is defined as the result of merging M_i with every $\frac{1}{\alpha}$th element of M_{i+1}.

As we're taking $\alpha = \frac{1}{2}$, M_i will be the result of merging M_i with every alternate element of M_{i+1}. For every **cascaded element** of M_i where i < k, we keep two pointers from each element which are derived as follows:

- If the element came from the same array, i.e, A_i, we keep a pointer to the two nearest neighbouring elements from M_{i+1}
- if the element has been cascaded, we keep a pointer to the predecessor of the element in M_i

These pointers helps to find the position of the query element x in A_i and also in the cascaded arrays below in $O(1)$ time.

Note: Since we are merging every alternate element of the below list to the current list, we have $|M_i| = |A_i| + \frac{1}{2}|M_{i+1}|$, which in turn ensures that $|A_i| \leq 2n = O(n)$.

After we perform the aforementioned pre-processing, querying x in all k lists is done as follows. First, we make a query for x in M_1 using a binary search in $O(\log n)$. Once we have found the position of x in M_1, we use the *cascaded pointers* to find the position of x in M_2. Generalising this step, once we found the position of x in M_i where i < k, we use the cascaded pointers to find the position of x in M_{i+1}.

To find the location in M_{i+1}, we find the *two neighbouring elements* in M_{i+1} that came from M_i using the pointers we had assigned during the pre-processing phase. Now, these elements will have *exactly one element* between them in M_{i+1}. **Therefore, to find the exact location in M_{i+1}, we just have to do a simple comparison with only the intermediate element**. This is the significance of taking $\alpha = \frac{1}{2}$ as we just have to perform only one comparison, which takes $O(1)$ time, and hence we can retrieve the location of x in A_i from its location in M_i again in $O(1)$ time.

Hence, the time to perform the preprocessing for fractional cascading is O(nk), the total search time per query is $O(k+\log n)$ and, the total time taken by the Fractional Cascading algorithm is $O(q(k+))$ for q queries, which is an astronomical improvement over the previous algorithms.

7 Applications

This technique has various applications in numerous fields.

1. Computational Geometry
    - Half-Range Plane Reporting
    - Explicit Searching
    - Point Location
2. Networks
    - Fast Packet filtering in internet routers.
    - data distribution and retrieval in sensor networks
3. Linear Range Queries
    - As an accompaniment to *Segment Trees*

References

[1] Wikipedia article on Fractional Cascading

[2] Practical Demo Application to demonstrate Fraction cascading

[3] Opengenus reference on Fraction Cascading

[4] An essay on fractional-cascading by arpit bhayani