A New Algorithm for Fast Data Searching – An Imagination based Indexed Data Searching Technique

Abhishek Verma
Student B.E(IT) Final Year, Gwalior Engineering College, Gwalior, Madhya Pradesh, INDIA

ABSTRACT
In this paper, I have proposed a fast searching algorithm for data stored on index bases. In general in most of the practical real time situations, it gives fast result for all elements belong to the set(array), which is always less than to Omega(log (n)) means takes less time in comparison to Binary search, B-Tree Search, AVL Tree Search etc., which takes Big Oh O( log (n)) and Omega(log(n)) time for half of the element in array. Here I uses an imaginary(self-assumed) array based on starting and last indexed element and total number of elements in an array, which is applied in formulas with element to be searched and it suggests an index for that data/element, which is mostly more near to that element in comparison to above compared algorithms.

Keywords— Fast Search, Imagination base searching, Index based searching.

I. INTRODUCTION
In present era, searching has become the most important part of computer systems. It is required to save time and processing with correct data retrieval. This algorithm is based on “Divide and Conquer” strategy, in which a function (AV Search) is called which takes sorted array(say arr[]) as an input with higher index(say H) and lower index(say L) as a parameter with the data element to be searched. As all most of the algorithms, whose time complexity is based on the number of elements present in the array, but this algorithm uses the elements value for data to search in the elements stored in an array. Due to the dependency of the element value this algorithm can guess the more accurate location of values. This algorithm can be used at any place where data is stored on index bases like in array with the arrangement like ascending or descending manner.

This technique can be assumed as, to find the highest possibility of the data element to be present on that location. This algorithm has more advantages then other, but with only one disadvantage that its time complexity is exactly half of O(n) or O(n/2) for range of values (-∞ to -∞ + (n/2)) to (+∞ - (n/2) to +∞) where n is the number of elements (more discussed in “Time Complexity of AV Search Worst Case” Section).

Due to simplicity of this algorithm it is easy to understand the working of this algorithm by any person and easy to implement in any suitable area(which is indexed based data storing platform).

II. WORKING PRINCIPLE OF THIS NEW ALGORITHM (AV SEARCH)

This algorithm is based on “Divide And Conquer” strategy, in which a function (AV Search) is called which takes sorted array(say arr[]) as an input with higher index(say H) and lower index(say L) as a parameter with the data element to be searched(say Alpha). After this it can be further divided in to two parts to understand it easily.

Firstly, it imagines an array (say img[]) on the bases of first element, last element and total number of elements present in that array (arr). This imaginary array is not stored in memory anywhere and it is just a concept to understand it’s working.

Secondly, it search for the index of element (Alpha) in an array (img) and find the nearest or equal value of Alpha in img, because this array (img) is totally created on the bases of actual array (arr). This imaginary array is not stored in memory anywhere and it is just a concept to understand it’s working.

Secondly, it search for the index of element (Alpha) in an array (img) and find the nearest or equal value of Alpha in img, because this array (img) is totally created on the bases of actual array (arr). It is easy to create similar array (img)as of array (arr) (may or may not be equals to array(arr)).

Now, this resulted index(say X) is used to search for element (alpha) in real array (arr). If it matches then it returns the “Element Found” message and if it is not found then it compares that the element is higher than that of element in actual array(arr[X]),if yes then it again calls the function (AV Search) recursively with old array(arr), old
higher index(H) but with new lower index(say L which is now equals to X+1).

And similarly for second condition that if element(alpha) is smaller than that of element in actual array(arr[X]) if yes, then it again calls the function recursively with old array(arr),old lower index (L) but with new higher index(say H which is now equals to X-1). This process repeated continuously until the actual element is not found. Due to this imaginary array(img), it is easy to identify the possible nearest location of element and due to which it’s time complexity is less than that of O(log(n)) and almost half too, as its time complexity.

The above statements are suitable for the condition when a data element is present in array (arr), but when there is no similar data element in array (arr), then we can identify it by checking the resulted index (X) is between lower index (L) and higher index (H), otherwise return the message that the “Element Not Found” which is not in array.

III. ALGORITHM OF AV SEARCH (ABHISHEKVERMA SEARCHALGO.)

Algo : AV(arr, L, H, ∝)
{
    Avg_diff = \frac{arr[H]−arr[L]}{H−L}
    X = \frac{X−arr[L]}{Avg_diff}+L

    if(X > H || X < L) then return “Element Not Found”
    else if(arr[X] = = ∝) then return “Element Found at Location X”
    else if(arr[X] < ∝ ) then call AV(arr, X+1, H, ∝ );
    else if(arr[X] > ∝ ) then call AV(arr, L, X-1, ∝ );
}
END ALGO

Where,
AV is Abhishek Verma,
arr is an array to be passed as argument,
L is the lower index of array (arr),
H is the higher index of array (arr),
Avg_diff is Average Difference,
Symbol ∝ is the data element to be searched and
X is the suggested location.
As you can see two formulas Average Difference and a formula to find Index. Now let us take these formulas into action and observe there working.

Average Difference:
It is the main formula which is used to generate an imaginary array (img), the img is just a concept to understand the working of formula easily.
Let us consider an array arr[L,N] of size 10 with lowest index ‘0’ and highest index ‘9’. With the elements 3,8,12,15,19,23,30,39,42,48 which are in ascending order with no sequence.
arr[0,10] = {3|8|12|15|19|23|30|39|42|48}
Image shown below shows the concept of Avg.diff. ”
Finding Index:

Now, with this Average Difference (avg_diff) we can generate an imaginary array (say img[L,N]). Where L is lowest index and N is the number of elements.

\[ \text{img}[0,10] = (3|8|13|18|23|28|33|38|43|48) \]

This img array is generated by take starting element arr[L] “3” and make an A.P. (Arithmetic Progression) until arr[H] “48” data element. By taking difference as Avg_diff.

Hence, above mentioned array img is the array which can be generated by the Average difference value, and now we can search for the data.

Suppose, we need to search for Data Element ‘42’, and in order to find the index we use the imaginary array and find the nearest value and find it’s index.

As we can see 43 is the most nearest value to 42 and its index is ‘8’ (Say X). Hence, now we compare for 42 in array arr[] with index X.

As a result we see arr[X] = arr[8] = 42 and which is equals to the required element.

**NOTE:** Now, as you can see that if we need to find the possible nearest value in this imaginary array (img), then we need to make another algorithm which is another big task, but we can replace searching nearest element in imaginary array algorithm with a simple formula.

\[ X = \left\lceil \frac{\infty - \text{arr}[L]}{\text{Avg_diff}} \right\rceil + L \]

where, \( \infty \) is element to be searched,

L is the lower index passed, and

X is the assumed location.

By using this formula we can see that \( X = \left\lceil \frac{42 - 3}{5} \right\rceil + 0 = \left\lceil 7.8 \right\rceil = 8 \)

Usually, we take sealing of both the formula.

**Time Complexity of AV Search Algorithm**

Now, let consider the Time Complexity of the AV Search algorithm, it is little hard to find the time complexity because time consumed to search an element is not fully dependent to the number of inputs,because it is based on the value of element is inserted in array.Let us take a look how it is not fully depending on total number of inputs and find the Best, Average and Worst Case time complexity.

**Best Case:**

In this time required to find the location of element is always constant, which is in almost all the
algorithm, but one thing new in this algorithm is that, for the other algorithm Best Case only exists for only single elements in an array, but in this Best Case can exist for all the elements present in array.

\[ O(1) \]

Best case only exists when two case are satisfied which are:

1. Average difference is equals to the lowest difference between two successive elements in array “arr”.
2. Average difference is equals to the highest difference between two successive elements.

It means that each two successive elements having difference (say d) which is equals to Average Difference.

\[ d = \text{avg.diff} \]

**Average Case**

In average case time required to find the location of element is always less than to \( \log_2(N) \), where \( N \) is the total number of element. In general we can say it as \( \Omega(\log(N)) \). Here, we also have one advantage over other algorithm which is that all the other algorithm listed in Abstract have exactly \( \log(N) \) complexity for exactly half of the elements in array, but this algorithm don’t.

Time Complexity < \( \Omega(\log(N)) \)

\[
arr[20] = \{-\infty + 20, -\infty + 19, -\infty + 18 \ldots, -\infty + 10, +\infty - 10, +\infty - 11, \ldots, +\infty - 20\}
\]

As shown above, 20 is the number of elements having any lowest index and highest index.

**Worst Case**

In this time required to find the location of element is less than or equal to the half of the number of input which only exist for exactly two or less elements of array.

\[ O(N/2) \]

This complexity is for maximum two elements in that array. Not for all the other elements. For other element it will be lesser.

To imagine the situation let us consider the array given below:-

**Examples for Best, Average and Worst Case**

Shown below are the examples which satisfy all the conditions and gives the appropriate results which are required for easy understanding of the working of algorithm.

**Example for Best Case**

Consider the array “arr[L,N]” where \( L \) is lowest index and \( N \) is number of elements.

\[
arr[1,10] = \{5,11,17,23,29,35,41,47,53,59\};
\]

Where,

\( L = 1 \) (lowest index),
\( N = 10 \) (number of element) and
\( H = ? \)

\[ H = (N + L) - 1 = (10 + 1) - 1 = 10 \]

Let we search for \( \propto = 11 \) in array

Now, call the function AV Search with parameters \( arr, L=1, H=10, \propto = 11 \)

AV(arr,1,10,11)
Now, we find the value of Avg.diff.

\[
\text{Avg\_diff} = \frac{(arr[H] - arr[L])}{H - L} = \frac{59 - 5}{10 - 1} = \frac{54}{9} = 6
\]

Now, we find the index where it can exist in imaginary array.

\[
X = \left\lfloor \frac{\propto - arr[L]}{\text{Avg\_diff}} \right\rfloor + L = \left\lfloor \frac{11 - 5}{6} \right\rfloor + 1 = \left\lfloor \frac{6}{6} \right\rfloor + 1 = 1 + 1 = 2
\]

As a result X=2, we can easily see that the data element 11 is actually exists at arr[2]. Hence, we found the \( \propto \) in single hit.

Now if we consider it for any element like 23, 41, 47, etc., it only takes O(1) for all the elements in array “arr”.

\[
arr[0,20] = \{0,9,15,21,23,28,32,41,47,48,50,52,56,66,77,81,86,89,99,110\};
\]

Where,
- L = 0 (lowest index),
- N = 20 (number of element) and
- H = ?

\[H = (N + L) - 1 = (20 + 0) - 1 = 19\]

Let search for \( \propto = 77 \) in array

Now, call the function AV Search with parameters arr, L=0, H=19, \( \propto = 77 \)

\[\text{AV(arr,0,19,77)}\]

Now, find the value of Avg.diff.

\[
\text{Avg\_diff} = \frac{(arr[H] - arr[L])}{H - L} = \left\lfloor \frac{110 - 0}{19 - 0} \right\rfloor = \left\lfloor \frac{110}{19} \right\rfloor = [5.789] = 6
\]

Now, we find the index where it can present in imaginary array.

\[
X = \left\lfloor \frac{\propto - arr[L]}{\text{Avg\_diff}} \right\rfloor + L = \left\lfloor \frac{77 - 0}{6} \right\rfloor + 0 = \left\lfloor \frac{77}{6} \right\rfloor + 0 = [12.833] = 13
\]

As a result X=13, we can see that “arr[13]” is 66 and it is less than 77. So, we call function again but with one new parameter value of “L”

Hence, we call AV Search again as:

\[\text{AV(arr,X+1,H, } \propto)\]

It looks like this now,

\[\text{AV(arr,14,19,77)}\]

Now again we find Avg.diff., which is,

\[\text{Avg\_diff} = [5.5] = 6\]

And, with this Avg.diff. we find value of X which is,

\[
X = \left\lfloor \frac{77 - 77}{6} \right\rfloor + L = 0 + 14 = 14
\]

Now, new value of X is 14 and we see for “arr[14]” which is 77.

Hence, we find the data element 77.
Example for Worst Case

Consider the array “arr[L,N]” where L is lowest index and N is number of elements.

\[ arr[0,20] = \{-10000, -9999, -9998, -9997, -9996, -9995, -9994, -9992, -9991, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 10000\}; \]

As we know that for any worst case complexity is always O(N/2) for one or two elements only.

Here, “9991” is the only element whose time complexity is N/2 means this searches 9 elements before visiting the actual element. For this I prepare a table by which you can see the values of Average Difference and X(index assumed) by AV Search algorithm with values of L and H.

As we all know “arr” and \( \propto \) value will never going to change in process.

So, let we call "AV(arr,0,19,9991)"

<table>
<thead>
<tr>
<th>No. of Recursion</th>
<th>L If ( \propto \gg arr[X] ) then L=X+1</th>
<th>H If ( \propto \ll arr[X] ) then H=X-1</th>
<th>Avg_diff</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>19</td>
<td>1052.63=1053</td>
<td>18.9848=19</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>18</td>
<td>1176.35=1177</td>
<td>17.9838=18</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>17</td>
<td>1333.06=1334</td>
<td>16.9842=17</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>16</td>
<td>1538=1538</td>
<td>15.9960=16</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>15</td>
<td>1817.45=1818</td>
<td>14.9939=15</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>14</td>
<td>2221.11=2222</td>
<td>13.9945=14</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>13</td>
<td>2855.42=2856</td>
<td>12.9975=13</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>12</td>
<td>3997=3997</td>
<td>11.9994=12</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>11</td>
<td>6661.33=6662</td>
<td>10.9995=11</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>19982=19982</td>
<td>10=10</td>
</tr>
</tbody>
</table>

Now, we can see that \( \propto \) is found at last X which is 10 and it is at 10th number of recursive function call.

In short I’ll tell you the number of recursion of all the elements.

- 10000 takes only one function call, -9999 also takes only 1, -9998 calls AV function 2 times, -9997 also takes 2 calls, -9996 takes 3 calls, -9995 takes 3 calls, -9994 takes 4 calls, -9993 takes 4 calls, -9992 takes 4 calls, -9991 takes 5 calls of AV Search, +9991 calls AV search 10 times as we saw above, +9992 calls AV Search 9 times, +9993 calls 8 times, +9994 calls 7 times, 9995 calls 6 times,9996 calls 5 times,9997 calls 4 times,9998 calls 3 times,9999 calls 2 times,10000 calls AV function only 1 time.

**Search for Negative Resulting Data Element / “Element Not Found”**

Consider the array “arr[L,N]” where L is lowest index and N is number of elements.

\[ arr[1,10] = \{3,8,12,15,19,23,30,39,42,48\} \]

Here, it is now clear that L=1 and H=10

Let us search the out bound element like “50” it not exists in array and also greater than the highest element.

For \( \propto = 50 \),

We call AV Search as,

\[ AV(arr,1,10,50) \]

As a result,

Now we return “negative acknowledgment” that “Element Not Found” for this. Let us consider the example taken in Best, Average and Worst Case.

Before solving those examples let us understand that there will be two cases exists:

1. Element is greater than the highest element in array or smaller than the lowest element.
2. Element is lesser than the highest element in array and greater than the lowest element.

**Case 1**:

Element is greater than the highest element in array or smaller than the lowest element in array.
Avg_diff = \left\lceil \frac{48-3}{9} \right\rceil = [5] = 5

Now, we find X

X = \left\lceil \frac{50-3}{5} \right\rceil + 1 = [9.4] + 1 = 11

As we can see X is 11, and we know that 11 is not the index which is in between or equals to 1 to 10.

Let us search for element \( x = 1 \) which is lower than lowest element “3” in array “arr”.

By applying the formula we find the X as -1 which also not in range of array.

Hence, we return that “Element Not Found”, and Time complexity for non-data element in array and which is outbound it gives O(1) every time.

**Case 2:** Element is lesser than the highest element in array and greater than the lowest element.

**Consider Best Case Example:**

If you run this algorithm by yourself you can find that any inbound element for any best case array (best case is for successful searching array for each element) it also calls AV Search function but for 2 times only means its time complexity will always be O(1) for any size of array it calls this recursive function only two times.

Assume that if you have Lacks of data entry and you searched for non-existing data element then you can find that it always tells you that “Element Not Found” only at 2nd call of AV Search.

But, let us search for element \( x = 13 \) in our case.

As you know it produces,

Avg_diff = 6

\[ X = \left\lceil \frac{13 - 5}{6} \right\rceil + 1 = [1.833] + 1 = 3 \]

As a result X=3,

We know that,

\( x = 13 \) and

\( arr[x] = arr[3] = 17 \)

Here, \( x < arr[3] \)

This means we call AV Search again with \( H = 3 - 1 = 2 \),

AV(arr,1,2,13)

Now,

Avg_diff = 6

Now we can find X,

\[ X = \left\lceil \frac{13 - 5}{6} \right\rceil + 1 = [1.833] + 1 = 3 \]

Now, we can see this clearly that X is equals to “3” which is greater than H and

Hence, it returns “Element Not Found”.

As you can see it only takes two calls to function AV Search.

**Consider Average Case Example:**

For an average case time complexity will be \( O(\log(n)) \) or we can say that, the maximum time required to find that element is not in array takes one more AV Search call as the maximum number of AV Search called for successful “Element Found”. We can understand it as

\[ \text{Time required to say “Element not Fond”} = \Omega(\log(n)) + 1 \]

Where,

\( \Omega(\log(n)) \) is the time complexity for successful search.

Let us find \( x = 19 \) in same array discussed above in Average Case Example.

We know that for first AV Search Call it gives,

Avg_diff = 6

For this X for \( x \) will be,
\[ X = \left\lceil \frac{19 - 0}{6} \right\rceil + 0 = \left\lceil 3.166 \right\rceil = 4 \]

For \( X=4 \),
Now, we call AV Search with \( H=X-1=4-1=3 \) and with other as same values:

AV(\( arr,0,3,19 \))

For this,
\[ \text{Avg\_diff} = 7 \text{ and } X = \left\lceil \frac{19 - 0}{7} \right\rceil + 0 = \left\lceil 2.71 \right\rceil = 3 \]
Now, \( X=3 \) for this
\[ arr[x]=arr[3]=21 \text{ and } \propto < arr[3], \]
Now, we call AV Search with \( H=X-1=3-1=2 \) and with other as same values:

AV(\( arr,0,2,19 \))

For this,
\[ \text{Avg\_diff} = 8 \]
\[ X = \left\lceil \frac{19 - 0}{8} \right\rceil + 0 = \left\lceil 2.37 \right\rceil = 3 \]

Now, we can see this clearly that \( X \) is equals to “3” which is greater than \( H \) and
Hence, it returns “Element Not Found”.
Here, it takes 3 call of function AV Search.
Which is equals to \([\log(N)+1]\)

**Consider Worst Case Example**

For this time complexity will be exactly \([O(N/2)+1]\) where, \( O(N/2) \) is the time complexity of the data element found for worst case array arrangement.
Now taking no more time of yours, let see it.
Let us find \( \propto = 9989 \) in same array discussed above in Worst Case Example.

Algo :AV(\( arr, L, H, \propto \))

\{
\text{Avg\_diff} = \left\lfloor \frac{arr[H]-arr[L]}{H-L} \right\rfloor \\
X = \left\lceil \frac{X-arr[L]}{\text{Avg\_diff}} \right\rceil + L \\
\text{if}(X > H \| X < L) \text{ then return “Element Not Found”} \\
\text{else if}(arr[X] = = \propto) \text{ then return “Element Found at Location X”} \\
\text{//Code changes from here to end…} \\
\text{else if}(arr[H]===\text{element}) \text{ then return “Element Found at H”} \\
\text{else if}(arr[L]===\text{element}) \text{ then return “Element Found at L”} \\
\text{else if}(arr[X] < < \propto) \text{ then call AV(\( arr, X+1, H-1, \propto \))} \\
\text{else if}(arr[X] > > \propto) \text{ then call AV(\( arr, L+1, X-1, \propto \))} \\
\}

END ALGO

For the first time Avg\_diff. will be 1053 and suggest \( X \) as 19 and it calculate same as it searched for 9991. But in end when it reaches to \( H = 9^{th} \) then it calls AV Search again with \( L = 10^{th} \) index to search, But in \( 11^{th} \) Call it fails to find and return “Element Not Found”.
Here, it takes 11 call of function AV Search. Which is equals to \( O(N/2)+1 \).

**Modifications**

We can make some modifications in this algorithm by which we can reduce its Time Complexity from 0% to 40%.
Algorithm after modification will looks like this.
Pros and Cons
There are many advantages and less disadvantages of this algorithm.

IV. ADVANTAGES OF AV SEARCH ALGORITHM

1. The main advantage of this algorithm is that it’s Worst Case situation can never being generated in real life situations means it always gives result faster than all the other algorithm which are listed above in Abstract. And after the modification made as listed above, then this algorithm will become much faster.

2. In best case for successful search this algorithm says that “Each element in given array have time complexity equals to O(1)”, and as we see in other algorithm best case “Only exist for single element in array”.

3. In average case time complexity for this algorithm is always less than O(log(n)). This means it is faster than that of the other entire algorithm listed in Abstract.

4. For any data element which is not listed in array in best case it gives O(1) or we can say constant and on the other hand, the other entire algorithm listed in Abstract don’t have any best case for non-belonging data element instead there complexity is exactly equals to O(log(N)).

5. For any data element which is not listed in array or non-belonging data element in average case its Time Complexity is less than or equals to \( O(\log(N)) \).

6. It’s time complexity varies (for non-belonging elements set) on the given situation but all the other algorithm compulsory have \( O(\log(N)) \).

V. DISADVANTAGES OF AV SEARCH ALGORITHM

1. The main disadvantage is that is gives it’s Time Complexity \( O(N/2) \) for Worst Case, for any practically illogical real time situations as discussed in Worst Case Section.

VI. CONCLUSION

This algorithm is based on “divide and conquer” strategy, and this algorithm challenges all the algorithms like Binary Search, AVL Search Tree, B-Tree types algorithm and gives fast result for any data/element belongs to that array or not, for all the practical real time situations which are not illogical set of array.

As a result it can be used in place where index based search is required like Oracle. It can be used simultaneously in small or large database systems, but it gives more feasible results in large database.

It saves the processing time for system for non-belonging data search time because its search time for non-elements is less than other algorithms.

Hence, this algorithm fits in almost all the situations.

REFERENCES

[2] Data Structure & Algorithm (By : VBD publication) Print 2014