International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 11 | Nov 2020 www.irjet.net p-ISSN: 2395-0072

Median Deviation Array Searching Algorithm

Shansita Das Sharma[1]

[1]*Student, Class 11, Shikshantar School, Haryana, Gurgaon*

Abstract - *This paper proposes the concept of a new single dimensional array search algorithm using mean deviation of quartiles and extremes of a dataset about the median. It also highlights the algorithm's working principles, illustratively explains the algorithms and compares it with pre-established searching algorithms.*

***Key Words*: Median Deviation Search, Interpolation Search, Prediction, Quartiles, Extremes**

1. INTRODUCTION

Array searching is a commonly used function in most small and large programs to retrieve a specific item from a set [1]. There are several algorithms for searching for a value in an array, each having its own advantages and disadvantages depending on the distribution of values in the dataset [2]. There is always scope for new searching techniques based on the data available to be searched. Median Deviation Search is another new array searching algorithm, which is most efficient when the data is uniformly distributed.

Properties

This is an algorithm, that, given an input of an array with n numbers, finds the index that of a value searched for in the array. Input required:

1. An array of numbers sorted in increasing order
2. Value to be searched

It returns the index of the value in the array, or -1 if not found. If there are multiple indexes at which the value is found in the array, any one of the indices will be returned.

2. ALGORITHM

2.1 Working Principle

Similar to Interpolation Search, the Median Deviation Search algorithm works by attempting to predict the position of the searched value [3]. However the latter follows a different method – index of the value is predicted by using mean deviation of extremes and quartiles about the median. If the value is not found at that location, then the section of the array being searched is reduced at every step until the section contains only 5 values – the two extremes and the three quartiles of that section. These 5 values are then searched through linear search.

2.2 Working

Initially the section of the array under consideration is the entire array. So the array of reference points stores min, 1st quartile, median, 3rd quartile and max index of the entire array (note: the index numbers are stored, not the value at those indices).

points = {minimum, Q1, median, Q3, maximum}

points.length = 5

The algorithm uses the following formula to calculate mean deviation about the median of a set of values:

$$dev = \frac{\sum_{i=0}^{4} |p_i - x_m|}{n} \quad \text{.....(1)}$$

p_i = elements of points array

x_m = median

n = number of elements in the current section of the array arr

dev = mean deviation about the median

The predicted index of the value to be searched (val) is calculated using the formula:

$$loc = \frac{val - arr[min]}{dev} + min \quad \text{.....(2)}$$

loc = predicted index of val

val = value to be searched

min = minimum value of current section (min=points[0] initially)

dev = deviation calculated using equation(1)

As the program iterates, the section of arr array being traversed reduces by increasing min if val is above predicted index or decreasing max if val is below predicted index. Alternatively, if val is found at that index, then it is returned.

if val=arr[loc] -> return loc
if val<arr[loc] -> max = loc-1
if val>arr[loc] -> min = loc+1

The new section of array to be traversed is within the bounds of the new max and min (inclusive of max and min). Accordingly, the 1st quartile, median and 3rd quartile stored in points array are also changed. Deviation is calculated again for the new section using the new values in points array to achieve higher precision in predicting the index of val. At the time when the section of array being considered has fewer elements than points.length, the loop terminates and value is searched in the remaining section through linear search.

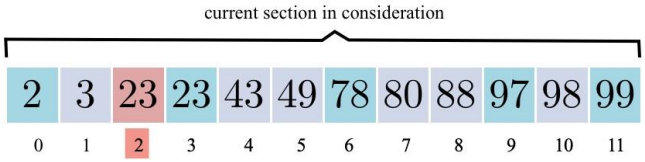
2.3 Algorithm Pseudocode

```
int update(int points[])
    int currLen = points[4]-points[0]+1 // section of array currently being considered
    points[1] = currLen/4 + points[0] //Q1
    points[2] = currLen/2 + points[0] //median
    points[3] = currLen*3/4 +points[0] //Q3
    return currLen

int medianDeviationSearch(int val, int arr[])
    // initialisations
    int arr_len = length of arr[]
    int points[] = {0, 0, 0, 0, arr_len-1}
    int p_len = 5 // length of points[]
    double dev = 0.0
    int loc = -1 // store predicted location of val
    int currLen = update(points) // store quartiles

    // repeat loop while currLen > points.length
    while currLen > p_len {

      if val>arr[points[4]] OR val<arr[points[0]]
      return -1

      // calculate deviation of current section
      for i=0 to i=4
          dev = dev + |arr[points[i]] - arr[points[2]]|
      dev = dev/currLen
      // predict index of val
      loc=(val-arr[points[0]])/dev + points[0]

      // check for value at predicted location
      if val equals arr[loc]
          return loc
      else if val < arr[loc]
          points[4] = loc-1
      else
          points[0] = loc+1

      currLen = update(points)
    }
    // search for value in points[]
    for i=0 to i=4
       if arr[points[i]] equals val
          return points[i]
    return -1
```

2.2 Illustrative Explanation

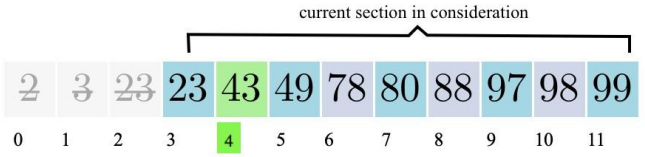
Let the value to be searched be = 43

Step I.



points = {0,3,6,9,11}
dev = 14.25
loc = 2
Value 43 not found at index 2 => repeat loop with min=loc+1

Step II.

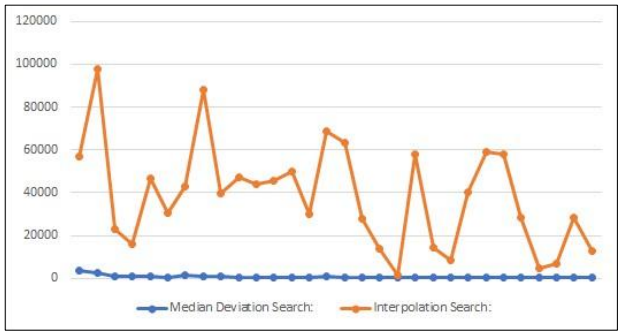


points = {3,5,7,9,11}
dev = 15.36111
loc = 4
Value 43 found at index 4 => Return 4

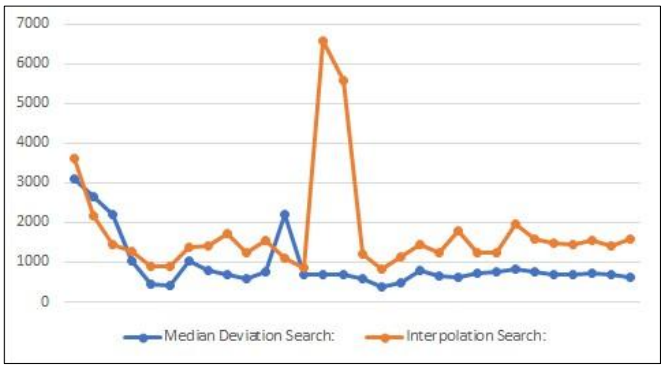
3. COMPARISON WITH INTERPOLATION SEARCH

In comparison to Interpolation Search, the probability of reaching the worst case scenario is significantly lower in Median Deviation Search for a large dataset with non-uniformly distributed data [4].

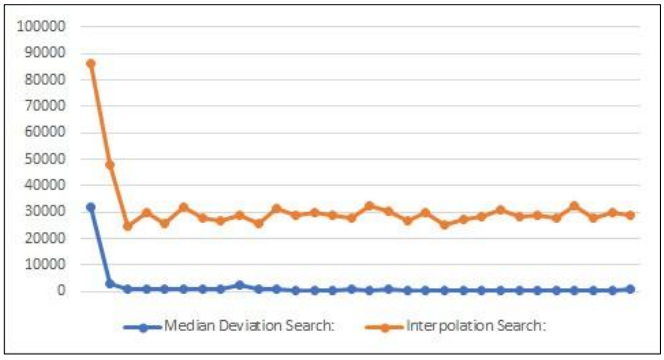
The following graphs compare runtimes of interpolation search and median deviation search for varying lengths of arrays. Each of the 30 points along the x-axis represents a unique array. In each array, 100 distinct values were searched. Y-axis shows average time taken in nanoseconds to search an element in each array.



Graph-1: Search times for arrays where array length is a random number between 1 and 10000



Graph-2: Search times for arrays where array length is 100



Graph-3: Search times for arrays where array length is 5000

4. CONCLUSION

This paper puts forward the Median Deviation Search algorithm to find the index of a value in a sorted array. Implementation of the algorithm and test results show that it is efficient in both uniform and non-uniform datasets. It is also observed that it is more efficient that existing search algorithms as the size of the dataset increases. Hence, it may have several applications where quick searching of data structures is required.

REFERENCES

1. Megharaja D.S, Rakshitha H J and Shwetha K, "Significance of Searching and Sorting in Data Structures," International Research Journal of Engineering and Technology, 2018.
2. Najma Sultana, Smita Paira, Sourabh Chandra. Sk Safikul Alam, "A Brief Study and Analysis Of Different Searching Algorithms," IEEE, 2017.
3. www.geeksforgeeks.org/interpolation-search/
4. Ahmad Shoaib Zia, "Title of paper with only first word capitalised," International Research Journal of Engineering and Technology, 2020.

APPENDIX
Reference Program in Java

```
public class medianSearchAlgorithm {

    int update(int points[]){
        int currLen = points[4]-points[0]+1; points[1]
        = currLen/4 + points[0]; //Q1 points[2] =
        currLen/2 + points[0]; //median points[3] =
        currLen*3/4 +points[0]; //Q3 return
        currLen;
    }

    int medianDeviationSearch(int val, int arr[]){
        // initialisations
        int arr_len = arr.length;
        int points[] = {0, 0, 0, 0, arr_len-1};
        int p_len = 5; // length of points[]
        double dev = 0.0;
        int loc = -1; // store predicted location of val
        int currLen = update(points); // store quartiles

        while (currLen > p_len) {

            if (val>arr[points[4]] || val<arr[points[0]])
                return -1;

            // calculate deviation of current section
            for (int i=0; i<=4; i++)
                dev+=Math.abs(arr[points[i]]-arr[points[2]]);
            dev = dev/currLen;
            // predict index of val
            loc=(int)((val-arr[points[0]])/dev) + points[0];

            // check for value at predicted location
            if (val == arr[loc])
                return loc;
            else if (val < arr[loc])
                points[4] = loc-1;
            else
                points[0] = loc+1;

            currLen = update(points);
        }

        // search for value in points[]
        for (int i=0; i<=4; i++)
            if (arr[points[i]] == val)
                return points[i];
        return -1;
    }
}
```