Distance Sort

Krishna Mohan Ankala *krishna.ankala@gmail.com*
Assoc.Prof, Dept of Computer science,
Ucek, JNTU Kakinada.

Hari Krishna Gurram, *harikrishna553@gmail.com*
M.Tech (CS),
Ucek, JNTU Kakinada.

Shanmukha Rao Kummari *kshanmuk@gmail.com*
M.Tech(CS),
Ucek, JNTU Kakinada.

Abstract

One of the fundamental issues in computer science is ordering a list of items. Although there is a number of sorting algorithms, sorting problem has attracted a great deal of research, because efficient sorting is important to optimize the use of other algorithms. This paper presents a new sorting algorithm which runs faster by decreasing the number of comparisons by taking some extra memory. In this algorithm we are using lists to sort the elements. This algorithm was analyzed, implemented and tested and the results are promising for a random data.

Keywords: Distance Sort, Distance

1. INTRODUCTION

Today real world getting tremendous amounts of data from various sources like data warehouse, data marts etc. To search for particular information we need to arrange this data in a sensible order. Many years ago, it was estimated that more than half the time on commercial computers was spent in sorting. Fortunately variety of sorting algorithms came into existence with different techniques [1].

Many algorithms are well known for sorting the unordered lists. Most important of them are merge sort, heap sort, shell sort and quick sort etc. [2]. As stated in [3], sorting has been considered as a fundamental problem in the study of algorithms, that due to many reasons:

- The need to sort the information is inherent in many applications.
- Algorithms often use sorting as a key subroutine.
- Many engineering issues come to the fore when implementing sorting algorithms.
- In algorithm design, there are many essential techniques represented in the body of sorting algorithms.
-

Sorting algorithms plays a vital role in various indexing techniques used in data warehousing, and daily transactions in online Transactional processing (OLTP). Efficient sorting is important to optimize the use of other sorting algorithms that require sorted lists correctly.

Sorting algorithms can be classified by:

- Computational complexity (best, average and worst behavior) of element comparisons in terms list size n. For a typical sorting algorithm best case, average case and worst case is O(n log n), example merge sort.
- Number of swaps
- Stability : A sorting algorithm is stable if whenever there are two records X and Y, with the same key and X appearing before Y in original list, X will be appear before Y in the sorted list.
- Usage of memory

In this paper, a new sorting algorithm (Distance sort) is proposed; here the basic idea is by taking a random sample of elements in the input we calculate the distance first, which is used to sort the elements in the given input. Here, we are taking the sample size as one percent of the total size of the input. As compared to other sorting algorithms this algorithm takes more memory, To restrict it from using very huge memory we are restricting the size of the list to 1.5 * n in sorting the data with n elements.

Section 2 presents the concept of Distance sorting algorithm and its pseudo code. Section 3 shows the implementation results for various sizes of random input. Finally, the conclusion was presented in section 4.

2: DISTANCE SORT

2.1 Concept

This algorithm works efficiently on random data by calculating the approximate position of the element. The main logic presented here is by calculating the approximate position we are able to minimize the number of comparisons, obviously the efficiency of the algorithm is increased.

2.2 Pseudocode

In pseudocode**,** the distance sort algorithm might be expressed as,

```
function sort ( input, size )
1.  var sizeOfSample := size / 100
2.  average( input )
3.  var maximum, minimum
4.  var distance := getDistance()
5.  if distance := 0
6.      distance := 1
7.  end if
8.  findMaxMin(input)
9.  marker = maximum + 1
10. var approxEle := getApproxEle() + 1
11. var constraintSize  = 1.5 * size
12. if approxEle > constraintSize
13.    approxEle := constraintSize
14. end if
15. Node in[approxEle]
16. initializeNode()
17. for i:=0 to size do
18.    var x := input[i]
19.   var position := ( x – minimum) /       distance
20. If position > approxEle
21.    Position := approxEle
22. end if
23. if in[position].element := marker
24.   in[position].element := x
25. end if
26. else
27.     if ( in[position].element >= x )
28.        Node temp
29.        temp.element = x
30.        temp.next = in[position]
31.        in[position] = temp
32.    end if
33.    else
34.      var flag := 1
35.      Node temp1 := in[position]
```

```
36.     Node temp := in[position]
37.      while (temp1.element < input[i])
38.          temp := temp1
39.          temp1 := temp1.next
40.          if( temp1 := null )
41.             temp1.element := x
42.             flag := 0
43.             break
44.          end if
45.       end while
46.    if flag := 0
47.       Node temp2
48.       temp2.element := input[i]
49.       temp.next := temp2
50.    end if
51.    else
52.     temp2.element := input[i]
53.    temp2.next := temp1
54.   temp.next := temp2
55.  end else
56.   end else
57.  end for
58.  var counter := 0
59.  for i:=0 to approxEle
60.     Node temp3 := in[i]
61.     while ( temp3.element != 0 )
62.         input[counter]:= temp3.element
63.        counter := counter + 1
64.        temp3 := temp3.next
65.        if temp3 := null
66.            break
67.       end if
68.       if (counter :=  (size -1) )
69.           break
70.      end if
71.    end while
72.    if ( counter := ( size -1 ) )
73.      Break
74.     end if
75.   end for
76.  end sort
```

Line 1, declares a variable sizeOfSample, which calculates the one percent of total input elements. By using this variable we are going ti calculate the approximate distance between the elements.
In line2, we called the function, *average*, the pseudocode for that function is given below.

```
function average(input)
1.  var counter := 0
2.  for counter 0 to sizeOfSample
3.    randPos := rand( size )
4.   average :=average +  input[randPos]
5.  End LOOP
6.  average = average/sizeOfSample
7.  end average
```

In line 4 of sort method we are calling the getDistance method which is used to calculate the approximation distance between the random elements, the pseudocode for the getDistance method is given below.

```
function getDistance()
```

```
1.  var dist =  (2*average)/ size
2.  if(dist < 0 )
3.     dist = -1 * dist
4.  Return dist
5.  end getDistance
```

In line8 of sort method we are calling the *findMaxMin* method, which is used to calculate the maximum and minimum values in the input elements.

```
function findMaxMin()
1.  var counter := 0
2.  maximum := input[0]
3.  minimum := input[0]
4.  for counter 1 to size
5.     if ( minimum > input[counter] )
6.        minimum := input[counter]
7.    end if
8.     if(maximum < input[counter] )
9.        maximum := input[counter]
10.   end if
11. end for
12. end findMaxMin
```

In line 9 of sort method we are calling the *getApproxEle* method which is used to calculate the approximate number of Nodes used to sort the given data. The pseudocode for the *getApproxEle* method is given below.

```
function getApproxEle()
1.  return (maximum – minimum)/distance
2.  end getApproxEle
```

Lines 10, 11, 12, 13 are used to constrain the number of nodes to sort the input elements. Line 14 declares an array *in* of type Node. The pseudo code for the structure Node is given below.

```
1.  struct Node
2.     int element
3.     Node next
4.  End struct Node
```

In line 15 of *sort* method we are calculating the *initializeNode* method which is used to initialize the nodes. The pseudocode for the initializeNode is given below.

```
Function initializeNode()
1.  var counter := 0
2.  for counter 0 to approxEle
3.     in[counter].element := marker
4.     In[counter].next := null
5.  end for
6.  end initializeNode
```

In line 18, we are calculating the approximate position for each input element, by calculating the approximate position we are going to reduce the number of comparisons. Lines 19 to 70 sort the elements by comparing from the approximation position.

3: IMPLEMENTATION RESULTS

Number Of Elements	Time Taken	Total comparisons
1000000	335	1428360
2000000	725	2856095
3000000	1105	4282827
4000000	1492	5708973
5000000	1874	7135946
6000000	2300	8558582
7000000	2666	9983612

TABLE 1: Best case scenario for Distance Sort (Time in milliseconds)

When there are very less number of duplicates in the input then this algorithm works in best case. If there are more duplicates in the input then this algorithm goes into the worst case.

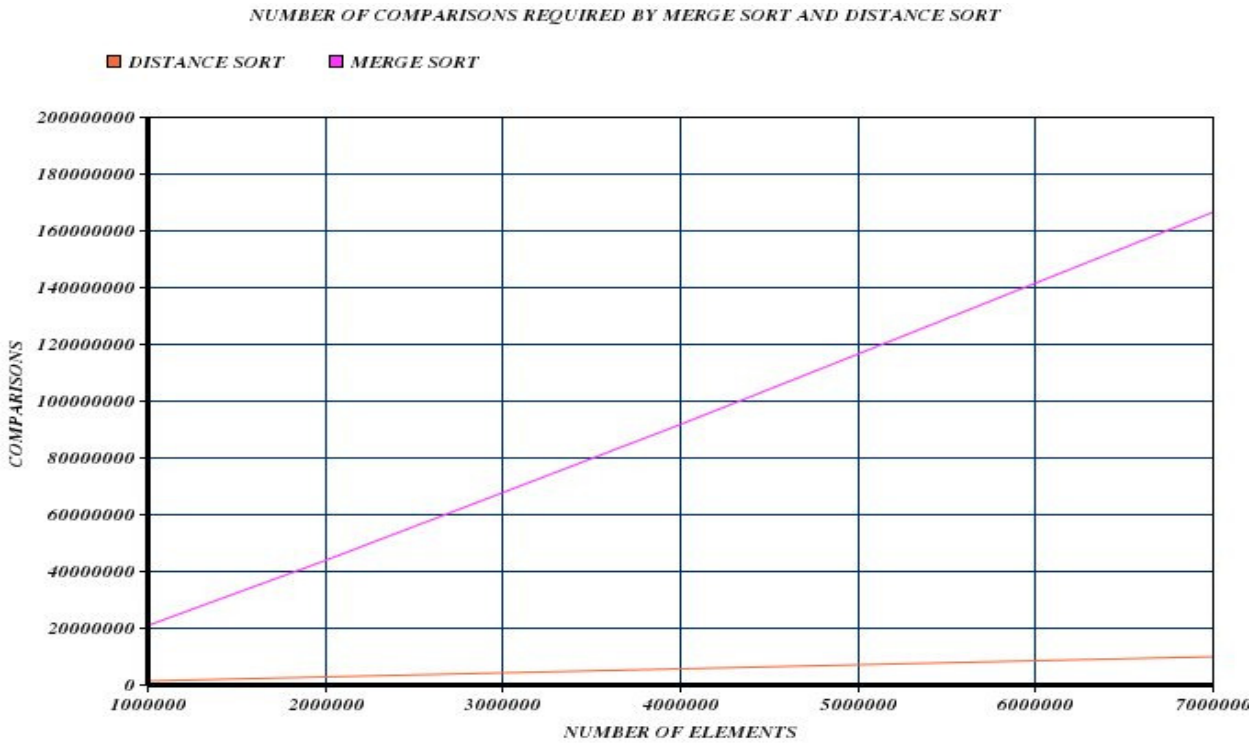

FIGURE 1: Best case comparison of Distance sort Vs Merge Sort.

Fig 1 shows that in best case distance sort work far better than the merge sort, Since in best case the position of an element is found approximately equal to the actual position. The best case occurs for the distance sort only when there are less number of duplicates.

Since we are finding the input element position approximately by the distance (we calculated from the average value), If the distance is calculated appropriately, then this algorithm works in best case, if the distance is not appropriate then this algorithm goes into worst case.

4: CONCLUSION

This distance sorting algorithm works very fast when there are very less number of duplicates in the input data and this algorithm totally depends on the distance value we are calculating to find out approximate position of the element to reduce comparisons.

5: REFERENCES

[1] Kruse R., and Ryba A., *Data Structures and Program Design in C++*, Prentice Hall, 1999.

[2] Shahzad B. and Afzal M., "Enhanced ShellSorting Algorithm," *Computer Journal of Enformatika*, vol. 21, no. 6, pp. 66-70, 2007.

[3] Cormen T., Leiserson C., Rivest R., and Stein C., *Introduction to Algorithms*, McGraw Hill, 2001.

[4] Aho A., Hopcroft J., and Ullman J., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[5] Astrachanm O., *Bubble Sort: An Archaeological Algorithmic Analysis*, Duk University, 2003.

[6] Bell D., "The Principles of Sorting," *Computer Journal of the Association for Computing Machinery*, vol. 1, no. 2, pp. 71-77, 1958.

[7] Box R. and Lacey S., "A Fast Easy Sort,"*Computer Journal of Byte Magazine*, vol. 16,no. 4, pp. 315-315, 1991.

.

[8] Deitel H. and Deitel P., *C++ How to Program*, Prentice Hall, 2001.

[9] Friend E., "Sorting on Electronic ComputerSystems," *Computer Journal of ACM*, vol. 3, no. 2, pp. 134-168, 1956.

[10] Knuth D., *The Art of Computer Programming*,Addison Wesley, 1998.

[11] Ledley R., *Programming and Utilizing Digital Computers*, McGraw Hill, 1962.

[12] Levitin A., *Introduction to the Design andAnalysis of Algorithms*, Addison Wesley, 2007.

[13] Nyhoff L., *An Introduction to Data Structures*, Nyhoff Publishers, Amsterdam, 2005.

[14] Organick E., *A FORTRAN Primer*, AddisonWesley, 1963.

[15] Pratt V., *Shellsort and Sorting Networks*,Garland Publishers, 1979.

[16] Sedgewick R., "Analysis of Shellsort andRelated Algorithms," *in Proceedings of the 4th Annual European Symposium on Algorithms*,pp. 1-11, 1996.

[17] Seward H., "Information Sorting in theApplication of Electronic Digital Computers to Business Operations," *Masters Thesis*, 1954.

[18] Shell D., "A High Speed Sorting Procedure,"*Computer Journal of Communications of the ACM*, vol. 2, no. 7, pp. 30-32, 1959.

[19] Thorup M., "Randomized Sorting in O(n log logn) Time and Linear Space Using Addition,Shift, and Bit Wise Boolean Operations,"*Computer Journal of Algorithms*, vol. 42, no. 2,pp. 205-230, 2002.