Journal of Al-Qadisiyah for computer science and mathematics
Vol.5 No.2 year 2013

Page78-88

Interpolation sort algorithm for lists that contain extreme values

Bahaa mohsen zbeel
College of fine art
Bahaamohsen95@yahoo.com

Recived : 4 /7 /2013 Revised: 4 /11 / 2013 Accepted: 6 /11 / 2013

Abstract

This paper describes a technique for applying interpolation sort algorithm on lists that contain extreme values. The traditional interpolation algorithm costs O(n) time and space complexity, where n is number of elements in the list, but with drawback of its limitation of application on just lists contain no extreme values. The proposed technique adapt the algorithm to sort list contain extreme values by reindex it and reformulate the linear interpolation formula.

1. introduction

Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised. Some of these algorithms are straightforward adaptations of schemes we use in everyday life. Others are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer. After years of study, there are still unsolved problems related to sorting. New algorithms are still being developed and refined for special purpose applications[Clifford 2013].

There are basically two kind of sorting algorithms, $O(n^2)$ and $O(nLog(n))$. $O(n^2)$ algorithms takes more time but less space while $O(nLog(n))$ algorithms takes less time but more space. So $O(n^2)$ algorithms are preferred for small arrays and $O(nLog(n))$ for large arrays There are sorting algorithms with $O(n)$ time complexity too like Radix sort but with limitation on the range of the data[Gourav 2012].

In this research, an adaptation for interpolation sort is proposed to sort lists that contain extreme values in $O(n)$ time and space complexity, so the resulted algorithm will be more efficient from data dependency perspective.

The rest of the paper is organized as follows: Section 2 provide brief overview of sorting algorithms terminology and their concepts. Section 3 present interpolation sort algorithm and discuss the proposed technique for applying interpolation sort algorithm on list contains extreme values with examples and analysis. Section 4 present conclusions.

2. Sorting terminology

Given a set of records $r_1, r_2, ..., r_n$ with key values $k_1, k_2, ..., k_n$, the Sorting Problem is to arrange the records into any order s such that records $r_{s1}, r_{s2}, ..., r_{sn}$ have keys obeying the property $k_{s1} \leq k_{s2} \leq \ldots \leq k_{sn}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order[Clifford 2013].

When duplicate key values are allowed, there might be an implicit ordering to the duplicates, typically based on their order of occurrence within the input. It might be desirable to maintain this initial ordering among duplicates. A sorting algorithm is said to be *stable* if it does not change the relative ordering of records with identical key values[Clifford 2013].

When comparing two sorting algorithms, the most straightforward approach would seem to be simply program both and measure their running times [Clifford 2013]. However, such a comparison can be misleading because the running time for many sorting algorithms depends on specifics of the input values. In particular, *the number of records*, *the size of the keys and the records*, *the allowable range of the key values*, and the amount by which the input records are "out of order" can all greatly affect the relative running times for sorting algorithms[Clifford 2013].

When analyzing sorting algorithms, it is traditional to measure *the number of comparisons* made between keys. This measure is usually closely related to the running time for the algorithm and has the advantage of being machine and datatype independent. However, in some cases records might be so large that their physical movement might take a significant fraction of the total running time. If so it might be appropriate to measure *the number of swap operations* performed by the algorithm. In most applications we can assume that all records and keys are of fixed length, and that a single comparison or a single swap operation requires a constant amount of time regardless of which keys are involved[Clifford 2013].

Some special situations "change the rules" for comparing sorting algorithms. For example, an application with records or keys having widely varying length (such as sorting a sequence of variable length strings) will benefit from a special-purpose sorting technique. Some applications require that a small number of records be sorted, but that the sort be performed frequently. An example would be an application that repeatedly sorts groups of five numbers. In such cases, the constants in the runtime equations that are usually ignored in an asymptotic analysis now become crucial[Clifford 2013].

comparison-based sorting algorithm An algorithm that makes ordering decisions only on the basis of comparisons[mark allen weiss 2010].

inversion A pair of elements that are out of order in an array. Used to measure unsortedness[mark allen weiss 2010].

selection The process of finding the kth smallest element of an array[mark allen weiss 2010].

lower bound of sorting The lower bound defines the best possible efficiency for any algorithm that solves the problem, including algorithms not yet invented. A simple estimate for a problem's lower bound can be obtained by measuring the size of the input that must be read and the output that must be written. Certainly no algorithm can be more efficient than the problem's I/O time. From this we see that the sorting problem cannot be solved by any algorithm in less than (n) time because it takes at least n steps to read and write the n values to be sorted. Alternatively, any sorting algorithm must at least look at every input vale to recognize whether the input values are in sort order. So, based on our current knowledge of sorting algorithms and the size of the input, we know that the problem of sorting is bounded by O(n) and O(n log n) [Clifford 2013].

We can prove a nontrivial lower bound for sorting. Our best upper bounds match the lower bound asymptotically, and so we know that our sorting algorithms are asymptotically optimal. Moreover, we can use the lower bound for sorting to prove lower bounds other problems[Thomas 2009].

3. The modified interpolation sort

In this section the traditional interpolation sort algorithm is described and then the modification is presented.

3.1 Interpolation sort

Consider the following unsorted array of size 15[Gourav 2012]:

56	32	12	65	37	80	55	60	40	77	50	9	68	35	20

The backbone of the sorting algorithm is the interpolation formula[Gourav 2012]:

$$IPOS[i] = SPOS + (N-1) \times \frac{(DATA[i] - DATA[MIN])}{(DATA[MAX] - DATA[MIN])}$$

where,

IPOS[i] → Interpolated position of the ith element of the unsorted array.
SPOS → Starting index of the array.
N → Number of elements in the array.
DATA[i] → Data at the ith position of the unsorted array
DATA[MIN] → Smallest data of the array.
DATA[MAX] → Largest data of the array.
It is to be noted that the division performed in the formula is integer division, i.e. decimal part is ignored.
For the given array,
SPOS=1
N=15
DATA[MAX]=80
DATA[MIN]=9

Substituting these values in the interpolation formula we get the interpolated positions of the elements as,

56	32	12	65	37	80	55	60	40	77	50	9	68	35	20	DATA
10	5	1	12	6	15	10	11	7	14	9	1	12	6	3	IPOS

Rearranging the array from smaller to bigger IPOS we get:

12	9	20	32	37	35	40	50	56	55	60	65	68	77	80	DATA
1	1	3	5	6	6	7	9	10	10	11	12	12	14	15	IPOS

So we see that most of the elements got sorted but there are few groups of elements whose IPOS values turned out to be the same. These groups of elements are treated as sub-arrays. The above technique is applied on each of these sub-arrays until we get no further sub-arrays [Gourav 2012].

It shows a high probability to show *O*(n) time complexity for a *well distributed* data. The algorithm has a disadvantage of large code size and taking a lot of RAM memory for sorting [Gourav 2012].

To achieve the same algorithm in reality we have to find a way to “rearrange the array from smaller to bigger IPOS”.Two structure types have to be used to do the above Effectively [Gourav 2012]. They are:

NODE1

DATA	RIEGHT

The data type of the field DATA is that of the data being sorted.
RIGHT is a pointer of type NODE1.

NODE2

SPOSS	N	START

START is a pointer of type NODE1.
SPOS and N are integer variables.

An array “BEG” of type NODE2 is used to store the information of the sub-arrays yet to be sorted. The starting index of the sub-array in the main array is stored in SPOS while N contains the number of elements in the sub-array. The information of the sub-array to be sorted is always present in BEG [1], i.e. the first element of BEG. Using the information of the sub-array present in BEG[1] the sub-array is sorted[Gourav 2012].

The following is The traditional interpolation sort algorithm [Gourav 2012]:

```
Input: Unsorted array ARRAY[] of size SIZE.
Output: Sorted array ARRAY[] of size SIZE.
INTERSORT (ARRAY[], SIZE)
1)  Make an array BEG of type NODE2 and size 1 using
    dynamic memory allocation.
2)  Set BEG [1].SPOS=1, BEG [1]. $N$=SIZE and BEG
    [1].START=NULL.
3)  Set NUM = 1.
4)  Repeat steps 5 to 14 while NUM $\neq 0$
5)  Traverse ARRAY from index BEG [1].SPOS to
    BEG[1].SPOS + BEG[1].$N$ – 1 and find the maximum
    and the minimum data. Save them as MAX and MIN   respectively.
6)  Make an array SUBARRAY of type NODE2 and size
```

```
      BEG [1]. $N$ using dynamic memory allocation . For each
      and every element of SUBARRAY initialize there fields
      $N$=-1 and START=NULL.
7)  Set A = 0.
8)  Traverse ARRAY from index BEG [1].SPOS to
      BEG[1].SPOS + BEG[1]. $N$ – 1 and for each and every
      data in this range:
        a) Interpolate the position IPOS of the data in
           SUBARRAY using interpolation formula. For
           interpolation take SPOS = 1, $N$ = BEG [1].$N$, DATA[MIN] = MIN and
           DATA[MAX]=MAX.

        b) Save the element in the memory location pointed by
           SUBARRAY [IPOS].START. To do this create a
           variable VAR of type NODE1. Set VAR.DATA = THE DATA,
           VAR.RIGHT =  SUBARRAY[IPOS].START.
           Then Point SUBARRAY[IPOS].START to VAR.
        c) If SUBARRAY[IPOS]. $N$ = -1 set it to 1. Else increase it by 1.
        d) If SUBARRAY[IPOS].N = 4 then increment A by 1.
9)  Set NUM = NUM + $A$ – 1.
10) Make an array NEWBEG of type NODE2 of size NUM
      using dynamic memory allocation.
11) Traverse SUBARRAY and for each and every element for which $N \neq -1$
      a) Set the field SPOS = BEG[1].SPOS for very first element for which $N \neq -1$.
         For other elements SPOS= TEMP.
      b) Set TEMP = SPOS + $N$.
      c) Copy all the data from the memory location pointed by START to the
         ARRAY in consecutive array indices starting from index SPOS in the actual
         array.
         Delete the memory locations pointed by START.
      d) If $N$<=3 then sort the data in ARRAY using bubble or insertion sort.
      e) If $N$>3 copy the element in NEWBEG.
12) Copy all the elements of BEG to NEWBEG except BEG[1].
13) Delete BEG and SUBARRAY.
14) Set NEWBEG as BEG.
```

3.2 The proposed modification of interpolation sort algorithm

Before proceeding in demonstrate the modification applied on interpolation sort to run with lists contain non uniform distributed elements values ,the reason that cause the interpolation sort algorithm not work properly on non uniform distributed values (list contains extreme values) must be explained. The following demonstrate this fact. Suppose the following list must be sorted:

11	9	10	8	13	12

When apply the interpolation formula with the following values:
n=6
SPOS=1
Data[min]=8
Data[max]=13

We get the following interpolation positions:

Data	11	9	10	8	13	12
Ipos	4	2	3	1	6	5

For the same list, except for the max element value that has chosen to be 18,we get the following interpolation positions for the element of the list:

Data	11	9	10	8	18	12
Ipos	2	1	2	1	6	3

and For the same list, except for the max element value that has chosen to be 23,we get the following interpolation positions for the element of the list:

Data	11	9	10	8	23	12
Ipos	2	1	1	1	6	2

Again For the same list, except for the max element value that has chosen to be 28,we get the following interpolation positions for the element of the list:

Data	11	9	10	8	28	12
Ipos	1	1	1	1	6	2

Repeat interpolation formula for same list, except for max element value that has been chosen to be 33; we get the following interpolation positions for the element of the list

Data	11	9	10	8	33	12
Ipos	1	1	1	1	6	1

When applying the algorithm on this list for different max values, the interpolated positions grouped in groups with same value whenever max value increased and become extreme value with respect for other values in the list.
Continuing increasing the value of max element will minimize the number of groups and increase the number of element in these groups until all the element of the list, except max element, will be in one group. this happen for the above list with max element of

value 33. in this situation the interpolation sort algorithm must be repeated for the elements with same interpolation positions values, i.e. for list with n-1 element.

This scenario may be repeated if the max element of (n-1) list has irregular value with respect to the rest of the list. The worse case here when the values of list element highly variant according to each other. This cause the repetition of the algorithem (n-2) times, 2 in this expression means insulation the min and max elements from the computation because they computed first time in the beginning of the algorithm.

Why this happened?

The answer of this question can be obtained from the interpolation formula that state:

The interpolated position($IPOS[i]$) of each element of the list is the difference between them and the min value element in the list $(DATA[i]-DATA[MIN])$ scaled by the ratio between the number of elements in the list, except max element, and the range of the element values of the list($\frac{(N-1)}{(DATA[MAX]-DATA[MIN])}$), with respect to the beginning location of the list, i.e. interpolated position offset or start position, $SPOS$.

The ratio in the Interpolation formula ($\frac{(N-1)}{(DATA[MAX]-DATA[MIN])}$) is responsible Of the failure of interpolation sort algorithm when max value be extreme with respect to the rest of the list because it puts more element values in same location in the resulted sorted list as it decrease when max value increase.

In the preceding list when max value was 13 the above ratio was 1 this mean each element in unsorted list occupy one distinct location in sorted list because the range of element values $(DATA[i]-DATA[MIN])$ of the list is equal to its length except max value $(N-1)$.

But when max value increase and become 18 the ratio become (1/2) this mean each two elements in unsorted list have successive values have same interpolation location in sorted list. 8 and 9 occupy same interpolated position in this list, this is an example.

In successive list with max values 23, 28 and 33 the ratio become (1/3),(1/4) and (1/5) respectively . That's means each 3 or 4 or 5 element have same interpolated position in sorted list, resulted in inefficient algorithm.

The interpolation sort will run inefficiently also when min value is extreme value with respect to the rest of the list elements values in same manner.

The proposed solution can be summarized using the word ' smoothing'. That's mean reindex the largest list element such that its value become its index ,so, no repeated positions result when applying linear interpolation formula to list elements.

The above procedure means as if the list is augmented by adding a uniform increasing values to smooth the difference between list elements. Starting from min value $DATA[MIN]$ and increase it by certain constant to obtain another value and increase it by same constant until $DATA[MAX]$ is reached. For the list:

11	9	10	8	33	12

The min value is 8, and the max value is 33,.if 2 is used as an incremental constant from min value to max value, the list will be (the original values are written in bold face font , and the adding uniform distributed values are written as italic font) :

8	9	10	11	12	*14*	*16*	*18*	*20*	*22*	*24*	*26*	*28*	*30*	*32*	33

Mathematically can represent the above procedure as following:
$DATA[MAX]= DATA[MIN]+CK$. ….(1)
Where:
C is the constant increment for the values generated from $DATA[MIN]$ to $DATA[MAX]$.
K is ther number of increments of C magnitude to ADD to $DATA[MIN]$ to obtain $DATA[MAX]$.
So, the list length(N) will be:
N=K+1. ……(2)

When substituting equations (1) and (2) in the Interpolation formula of section 2.3 the interpolation formula will be :

$$IPOS[i] = SPOS + (DATA[i] - DATA[MIN]) \times \frac{1}{C}. \qquad i=1..N.$$

The most important thing that must be decided is the chosen of a suitable value for C.

For the following list:

11	9	10	8	33	12

If c is chosen to be 1 then the interpolated positions of list elements values will be:

Data	11	9	10	8	33	12
Ipos	4	2	3	1	6	5

But, if 12 is replaced with 14 in the above list then the interpolated positions will be:

Data	11	9	10	8	33	14
Ipos	4	2	3	1	6	7

the interpolated position of 14 is 7 , and this position is out of list's positions range.

the following table shows the interpolated positions for the list [11,9,10,8,33,14] .

List element values

C	11	9	10	8	33	14
1	4	2	3	1	6	7
2	2	1	2	1	6	4
3	2	1	1	1	6	3
4	1	1	1	1	6	2
5	1	1	1	1	6	1

The choice of C=1 is perfect but its unsafe from (out of range) interpolated positions. The choice of c>2 result in groups of more than 2 numbers if these numbers are convergent in there values that need the rerun the interpolation formula and that require find max and min values for each group.

The use of c=2 is perfect , because if there is groups of two numbers resulted in same interpolated positions after applying the interpolation formula , the only effort needed to rearrange them in correct order is compute the interpolated position for just two elements in each groups. This prevent produce elements with same interpolated positions again as may happen in groups length greater than two.

For the list [11,9,10,8,33,14], when applying the modified interpolation formula proposed in this paper with *C*=2 the interpolated positions for the list will be:

Data	11	9	10	8	33	14
Ipos	2	1	2	1	6	4

After apply the modified interpolation formula for the groups of same interpolated positions as stated in [Gourav 2012], the following interpolated positions will result:

Data	11	9	10	8	33	14
Ipos	4	2	3	1	6	4

Applying the modified interpolation formula for the elements 11 and 14 that have same interpolated position will result the following interpolated positions:

Data	11	9	10	8	33	14
Ipos	4	2	3	1	6	5

intervals. This is the configuration of the list that make the algorithm run in its best case.

The modified interpolation sort algorithm will run in *Θ(N)* time complexity for list Data incremented regularly by 2 over the range of list data $(DATA[MAX]-DATA[MIN])$, this mean the algorithm's time complexity incremented linearly with the number of list element , therefore , the time complexity of the algorithm can enhanced significantly just when run on more sophistgated CPU and on shortest time fetch cycle memory machine .

For the list:

14	12	10	8	33	16

When smooth the list using *C*=2 the list seems as if its length augmented to the following list(the list is distributed over intervals with two values):

8	9	10	11	12	13	14	15	16	17	18	19	20	21
Interval 1		Interval 2		Interval 3		Interval 4		Interval 5		Interval 6		Interval 7	

22	23	24	25	26	27	28	29	30	31	32	33
Interval 8		Interval 9		Interval 10		Interval 11		Interval 12		Interval 13	

When applying the modified interpolation formula, the following Interpolated positions will result:

Data	14	12	10	8	33	16
Ipos	4	3	2	1	6	5

We can show that the interpolated positions are the same the interval positions the list element belong to and run in *Θ(N)* time complexity. In this paper The modified interpolation sort algorithm will run in *Θ(N)* time complexity for list Data incremented regularly by 2 over the range of list data $(DATA[MAX]-DATA[MIN])$. this mean the algorithm's time complexity incremented linearly with the number of list element , therefore , the time complexity of the algorithm can enhanced significantly just when run on more advanced CPU and on shortest time fetch cycle memory machine .

3.3 Modified interpolation sort algorithm drawbacks
the main drawbacks of the algorithm is produce (out of range) interpolated positions when list elements values belongs to non contiguous interval over list data range after smoothing it.
For the list:

Data	14	20	18	8	33	16
Ipos	4	8	7	1	6	5

The elements values 18,20 have out of range interpolated positions 7,8

4. Conclusion

The non-comparison sort algorithm approaches the lower bound of sorting more than the comparison based one this is due to its ability to determine the correct position of list element directly using mathematical formula without take into account other list elements as in comparison based algorithm. But, in the other hand, it's affected by the distribution nature of list elements value.

References

[1]Clifford A. Shaffer,2013" Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)", Dover Publications, page 231.

[2] Gourav Saha and S. Selvam Raju,2012" Interpolation Sort and Its Implementation with Strings" *International Journal of Computer Theory and Engineering, Vol. 4, No. 5,:772-776, page 772.*

[3] Mark allen weiss, 2010"Data Structures & Problem Solving Using Java" fourth edition, pearson-Addison-Wesley, page 384

[4]Thomas H. Cormen, Charles E. Leiserson , Ronald L. Rivest and Clifford Stein,2009 "Introduction to Algorithms" Third Edition, the MIT press, page 148.