International Journal of Scientific & Engineering Research, Volume 4, Issue 8, August 2013
ISSN 2229-5518

INTERRUPTION CONTROLLED OPTIMAL SORTING

Rathinasabapathy R and Chandrasekaran S

Abstract— There are numerous algorithms have been developed to sort a given set of elements. Generally, sorting algorithms are having a computational complexity of $O(n^2)$. During the process of sorting when there is an interrupt due to system failure, the algorithm abruptly ends. Then the algorithm usually requires to be started again from the beginning. Given the quadratic complexity, when the sorting process is interrupted, restarting from the beginning, will be time consuming. In this paper, a solution has been provided, by way of using an algorithm, which will not start from the beginning, instead it will continue from where the sorting algorithm was interrupted by storing the intermediate data in the secondary storage.

Index Terms— sorting, searching, complexity analysis, algorithm.

1 INTRODUCTION

Sorting is one of the important issues in computer science [1][2][3][4]. A lot of sorting algorithms has been developed to enhance the performance in terms of computational complexity [5], memory and related factors. Although sorting problem is one of the basic issues, useful and efficient sorting algorithms are still being developed [6]. Any sorting process rearranges a set of elements, objects, numbers, strings, attributes of a database, combinations of attributes of a database, either in ascending or descending order. The selection sort, insertion sort, bubble sort are some of the fundamental algorithms, which are comparison-based. They have the complexities in the order of $O(n^2)$. The other sort algorithms in comparison-based category such as Quicksort, heap sort, Merge sort, Shell Sort etc have their complexities in the order of $O(n \log_2 n)$ [7][8]. The sorting algorithms have gained wide applications and a lot of analyses [3] have been on the efficacy of the algorithms. There are a set of algorithms, which modifies, enhances, optimizes the existing algorithms and provides visualization on the working of sorting [9][10][11][12][13]. Usually all algorithms are expected to execute from starting to end. When interrupted, mostly the process starts its sorting operation once again from the first element in the unsorted list. The time consumption will be comparably more, when its operation is perturbed in mid way. To remedy this problem, a solution is provided in this paper, to sort the elements from the point of interruption. The point of interruption may occur anywhere in the algorithm. To implement the algorithm, it is required to identify where the sorting has been interrupted and start the process from there. On an experimental basis, the proposed technique is incorporated in the popular selection sort algorithm.

2 SELECTION SORT

The following is the selection sort algorithm to sort an array of n numbers in ascending order.

```
1)  Read a[i] //a[i] to be sorted in ascending order
2)  For (i=0;i<n-1;i++)
3)      For(j=i+1;j<n;j++)
4)      {
5)          If (a[i]>=a[j])
6)          {
7)              tmp=a[i];
8)              a[i]=a[j];
9)              a[j]=tmp;
10)         }
11)     }
12) }
```

It is a well known sorting algorithm having a complexity of $O(n^2)$. If the algorithm is interrupted in the middle, to start the algorithm from the interrupted point, it becomes necessary to split the process into various sub processes and for each sub process, the starting point and ending point is to be analyzed and identified. In this paper, a method has been provided so that it can proceed from the interrupted point. For this, the algorithm may be split into several stages. If the algorithm is interrupted in the middle, for example let it be between i^{th} stage and $(i+1)^{th}$ stage, the algorithm has been designed in such a way that the status of i^{th} stage is retrieved from back store and the algorithm continue to start as if it has completed i^{th} stage. The intermediate results have be to stored in a backup store. The algorithm has to be developed with proper cognizance , so that, if at all there is an interruption , the status of the algorithm at that instant such as what are the items need to be stored ? and from which position it needs to be started again ? and what are all the data structures to be kept in the backup store ?.

When analyzing the selection sort, it can be found that the elements are sorted from beginning to end by bringing the smallest element to the top one after another for each iteration. If it is done for n iterations, the sorting will be over. The problem of sorting may be considered as bringing each element to the top which is smallest of all the remaining elements and it is iterated. Bringing each element to the top involves more smaller sub steps, which may be ignored because it is contributing to the complexity which is less than O(n). In short, the problem of $O(n^2)$ complexity is divided into problem of n times the complexity of O(n). The problem is not further

divided.

The selection sort algorithm can be rewritten delineating the each sub process involved.

1) For(i=0;i<n;i++)
2) {
3) Find smallest i^{th} element in each pass
4) }

Suppose the system is interrupted in the middle during say i=k. That is, already up to i=k-1, the smallest elements are found in the array a[i]. The algorithm is to be continued from i=k even though it is doing the operations in the middle of i=k. So, the intermediate results at the end of i=k-1, should have been stored in the backup store, to continue it. That is, the array of numbers is sorted up to i equal to k-1. So, the array is split into two disjoint arrays as c[ ], a sorted array and d[ ], a unsorted array. Array c[ ] will contain sorted k elements of the given array, and array d[ ] will contain remaining elements of the array. It is enough if we sort the array d[ ] and append it with array c[ ].

3 PROPOSED ALGORITHM

For splitting the array and to store it in backup store, every time, an element is brought up to the top as smallest(for ascending order) element. The element just found and the remaining elements are to be split into two arrays and to be stored in a backup store. For the next iteration the file is to be written afresh with two elements as sorted and the remaining elements as unsorted. Proceeding this way, at any point of time, the file will contain sorted array and unsorted array with the information that the size of the array which has been sorted. So, a structure can be created as below to store the elements to be backed up.

```
Struct arraydata
{
        int k;   //upto which array is sorted
        int c[ ]  //a sorted array
        int d[ ]  //a unsorted array
}
```

From the above discussion, the given sorted array can be viewed as in array data structure, having value k as zero, array c[ ] as empty and array d[ ] as the whole unsorted array. During the execution of the program the value of k, array c[ ] and array d[ ] will go on changing and this structure has to be kept in a file. If we proceed to store the data for each iteration in a file, it will be time consuming. To alleviate this, the intermediate data is stored after a specified time interval that could be chosen by the user. It can further be modified to store intermediate data in two files in two different consecutive time intervals. This will facilitate us to get to the intermediate data even if one file is not accessible other file can be used to start the sorting process at a point at which the algorithm is interrupted. An algorithm is explained in the next section with the incorporation of the methods explained above.

4 SPLITTING THE ARRAY

The following is the algorithm to sort the given array of data. In this algorithm, the objects used and their relevance has been specified in Table 1.

Table 1. Objects used and their relevance

Obj	An instance of the structure array data
Obj.k	The index upto which the data is already sorted
Obj.c[ ]	An array of sorted data
Obj.d[ ]	An array of unsorted data
fs,fs1,fs2	File stream for a text data file
Readfs	A function to read obj from a file stream fs
Writefs	A function to write obj into a file stream fs

Algorithm1 : To sort a given array

```
1)  Start = 0 // If started from beginning
2)  If ( Start == 0 ) { fmode = 0; Start = 1; }
3)  Else // if started from interrupted state
4)  {
5)  fs = (file stream with latest time of creation(fs1,fs2));
6)  If (fs == fs1 ) { fmode = 0; } Else { fmode = 1; }
7)  }
8)  Readfs(fs,obj)
9)  Start_time = time();
10)  szeofd = sizeof(d);
11)  For(p=0; p<szeofd-1;p++)
12)  {
13)   For(q=p+1; q<sizeofd; q++)
14)   {
15)      If(obj.d [p]>obj.d[q])
16)      {
17)              tmp=obj.d[p];
18)              Obj.d[p]=obj.d[q];
19)              Obj.d[q]=tmp;
20)      }
21)   }//end of for(q = …)
22)   End_time = time();
23)   Time_Elapsed = (End_time – Start_time)
24)   If ( Elapsed_Time() >=  Tolarable_time)
25)     { //append array c[ ] with array d[0..p ]
26)              fmode = (fmode+1) mod 2;
27)              If ( fmode == 0)
28)                    { Writefs(fs1,obj); }
29)              Else { Writefs(fs2,obj); }
30)              Start_time=time();
31)       }
32)   }//end of for(p = …)
```

In the initial stage, the algorithm starts with one file with the given data. As, the program is being processed, after an

elapsed time, a file with file stream fs2 is created. Then, again after an elapsed time the state of the system is stored in fs1. Alternatively, the files with streams fs1 and fs2 are created. If interruption occurs, the algorithm again started by taking the information in the file with latest time of creation. If by chance, it is not able to be used, then the previous state of the system is taken from the other file.

Instead of using the time-sliced approach as discussed earlier, an alternate algorithm has been proposed by setting the number of iterations to be executed before storing the intermediate sorted data. After completion of the preset number of iterations, the intermediate results are stored and the algorithm can continue execution from the point it was preset(that is from next iteration onwards) up to end of data.

The following algorithm assumes a file having the structure as below.

```
Struct arraydata
{
        string strng;
        string nm[ ]; // array of numbers
}
```

Each line of the file stores one value of the variables, strng or the array nm[i]. The variable "strng" stores the value which specifies the record number upto which the array of numbers are sorted. In this algorithm two files with the above structures are used. Initially, a file with file pointer [fp1] is storing the data with value -1 for the variable "strng" which specifies that the file is yet to be sorted. Once the algorithm started execution, after certain specified number of iterations, a file with file pointer [fp2] is stored with the number of records sorted in the variable "strng" and the intermediate data is written into the file. The two files are updated in this method with recent intermediate sorted data in an interleaved manner. If the algorithm abruptly ends in the middle, when the program is restarted again, the file having the greater value for the variable "strng" will have the latest sorted data. The algorithm will start with the latest updated data and do the same procedure until successfully ends with the result.

Algorithm2 : To Sort an array .

```
(1) open file with file pointer [ fp1 ].
(2) read first record and put it into [ strng ]
(3) convert [ strng ] to integer and store it into [ filestat1 ]
(4) if [ filestat1 > 0 ]
   then
   begin
    (4.1) open file with file pointer [ fp2 ]
    (4.2) read first record and put it into [ strng ]
    (4.3) convert [ strng ] to integer and store it into  [filestat2]
    (4.4) if [ filestat1 > filestat2 ]
            then
                begin
                        let [ fp ] be point to [fp1]
                        let [ rcdno ] be value of [ filestat1 ]
                        let [ prcd ]  as 1
                end
             else
                begin
                        let [ fp ] be point to [fp2]
                        let [ rcdno ] be value of [ filestat2 ].
                        let [ prcd ]  as 2
                end
          end
        else
          begin
                let [ fp ] be point to [fp1]
                let [ rcdno ] be -1
                let [ prcd ]  as 1
          end
(5) read the file pointed by [ fp ] and store the contents into array [ nm[ ]]
(6) let [ itrno ]  as 0 //representing iteration number 0
(7) for(i:[rcdno+1]..[len-1])
        (7.1) increment [ itrno ]
        (7.2) for(j:[i+1]..len)
              (7.2.1) if ( nm[i] > nm[j] )
                        { swap(nm[i],nm[j]) }  else  { }
        (7.3) If [ itrno > [tnoi] ]
                    //tnoi - tolerable_no_of_iterations
                    begin
                     if [ prcd == 1]
                            {
                            open file with pointer [fp2]
                            strng ← [ i ]
                            write [strng] into  file [fp2]
                            write array[ nm[ ] ] into file [fp2]
                            assign 2 to [ prcd ]
                            }
                     else
                            {
                            open file with pointer [fp1]
                            strng ← [i]
                            write [strng] into  file [fp1]
                            write array [nm[ ]] into file [fp1]
                            assign 1 to [ prcd ]
                            }
                            assign 0 to [itrno]
                      end

(8) print the array [ nm[ ] ]
```

5 CONCLUSION

A sorting algorithm with facilities to continue sorting from the position where it is interrupted, is proposed here. The interruption is expected to occur at any step during its execution. This algorithm uses two intermediate files to preserve the recent status of the sorting process. User can choose the time interval depending upon how much time it takes to sort and our tolerable time interval. One more algorithm has been proposed which stores intermediate data, after specified number of iterations in an interleaved manner. This concept can be applied to other existing sorting algorithms, as well as in other important and time consuming algorithmic applications. This technique may find its applications in parallel and distributed computing.

REFERENCES

[1] Horowitz, E.,Sahni. S, Fundamentals of Computer, Algorithms, Computer Science Press, Rockville. Md.
[2] Donald E. Knuth, The art of computer programming, volume 3: (2nd ed.) sorting and searching, 1998.
[3] V.A. Aho , J.E. Hopcroft., J.D. Ullman, The Design and Analysis of Computer Algorithms, second ed, Reading, MA:Addison-Wesley, 1974.
[4] Martin, W. A. Sorting. ACM Computing Surveys 3, 4 (1971), 147–174.
[5]Parag Bhalchandra, Nilesh Deshmukh, Sakharam Lokhande, Santosh Phulari, A Comprehensive Note on Complexity Issues in Sorting Algorithms, Advances in Computational Research, ISSN: 0975–3273, Volume 1, Issue 2, 2009, pp-1-09.
[6] Jehad Alnihoud and Rami Mansi, An Enhancement of Major Sorting Algorithms , The International Arab Journal of Information Technology, Vol. 7, No. 1, pp.55-62, January 2010
[7] Yingxu Wang, A New Sort Algorithm: Self-Indexed Sort, Communications of ACM SIGPALN, Vol.31, No.3, ACM, pp.28-36, March 1996
[8] C.A.R Hoare : Quicksort, Computer Journal vol 5(1962), 10-15
[9] Bremananth R, Radhika.V and Thenmozhi.S, Visualization of Searching and Sorting Algorithms, International Journal of Computer and Information Engineering 3:3 2009
[10]Jon L.Bentley, M.Douglas Mcilory, Engineering a Sort Function, Software-Practice and Experience, Vol. 23(11), pp.1249–1265, November 1993.
[11] Robert Lafore, Data Structures and Algorithmsin Java,Second Addition,2002.
[12] Robert Sedgewick, Algorithms in Java, Third ed. Parts 1-4, Addison Wesley, 2003
[13]Mark Allen Weiss: Datastructures & Algorithm analysis in Java, Addison Wesley, Reading Mass., 1999