Upgraded Selection Sort

Sunita Chand, Teshu Chaudhary, Rubina Parveen

Department of Computer Science & Engineering ,
Krishna Engineering College
95-Loni Road, Near Mohan Nagar , Ghaziabad, U.P., India

***Abstract*— Sorting is a technique which is frequently used in our day to day life. It has a direct implication on searching. If the data is sorted on any key attribute, finding data based on that key attribute becomes very fast. There are many sorting algorithm that are being used in practical life as well as in computation. We will concentrate on selection sort in this context that has already been used for so many years and are providing a solution to make it more faster as compared to previous one.**

The upgraded Selection sort works by repeatedly selecting the minimum or the maximum value and placing them in their proper position in the list. We provide a method of selecting both the minimum and the maximum value simultaneously and placing them in their respective positions in a single pass. so that the length of array reduces by two elements in each pass which reduces the number of passes by n/2.Thus array get sorted in ascending order from the beginning of the array, and in descending order from the end and hence the sorting is done from both the ends of the array and hence the array is sorted in less time using this upgraded selection sort as compared to the original selection sort. The complexity of the upgraded selection sort provided in this work comes out to be same as that of the original selection sort $O(n^2)$ but the total number of passes in the original selection sort is n and in upgraded algorithm is n/2.

Keywords-sorting; selection sort; minimum; maximum; swap; in place; algorithm

I. INTRODUCTION

A selection sort is one which successive elements are selected in order and placed into their proper sorted position. The straight selection sort is an **in-place sort** as it does not require the additional space to sort the array except a few temporary variables. The straight selection sort[1] consists entirely of a selection phase in which the largest of the remaining elements, large, is repeatedly placed in its proper position, i, at the end of the array. To do so, large is interchanged with the element x[i]. The initial n-element array is reduced by one element after each selection. After n-1 selections the entire array is sorted. Thus the selection process need be done only from n-1 down to 1 rather than down to 0.
The following C function implements straight selection:

```
. void SelectSort(int x[], int n)
{
        int  i, indx, j, large;

        for (i = n-1; i>0; i--)
        {
                /* place thr largest number of x[0] through*/
                /* a[i] into large and its index into indx */
                large = x[o];
                indx = 0;
                for(j = 1; j <= i; j++ )
                {
                        if ( x[j] > large )
                        {
                                large = x[j];
                                indx = j;
                        }/* end for …… */
```

```
                    x[indx] = x[i];
                    x[i] = large;
                } /*end for */
        }/*end for */
}/* end  SelectSort */
```

II. ANALYSIS OF THE STRAIGHT SELECTION SORT

Analysis of the straight selection sort is straightforward[1]. The first pass makes n-1 comparisons; the second pass makes n-2, and so on. Therefore, there is a total of

(n-1) + (n-2) + (n-3) + …….. + 1 = n * (n-1)/2

Comparisons, which is $O(n^2)$. The number of interchanges is always n-1 (unless a test is added to prevent the interchanging of an element with itself). There is little additional storage required (except to hold a few temporary variables). The sort may therefore be categorized as $O(n^2)$. The selection sort can be used for any file for which n is small.

III. UPGRADED SELECTION SORT

Here we present the new version of selection sort which works as follows:

Let us take an input array x with n elements. We find the minimum and the maximum elements of the array in first pass and interchange them with first i.e,x[0] and the last element i.e., x[n-1] respectively. In the next pass , the array is reduced by two elements. This time we start with the array starting with second element i.e x[1] and ending at second last element .e.g. x[n-2]. This way the array gets sorted from the beginning in the ascending order and from the end in descending order simultaneously. So the array will get sorted in n/2 passes.

The algorithm for the revised selection sort works as follows:The proposed algorithm has two functions:

Upgraded_Selection_Sort(int [], int) and Swap(int [], int, int).

First function **Upgraded_Selection_Sort(int** [],**int** n **)** takes two arguments an array and the length of the array .It uses the following variables for the purpose written in front of them :

Loc_Min: the indices of minimum elements of the array
Loc_Max: the indices of maximum elements of the array

1. We start the algorithm by setting both Loc_Min and Loc_Max to the initial element of the array say at ith position, at the beginning of each pass. i.e. Loc_Min= Loc_Max=i
2. Starting with the next element, every element is compared with the current value of Loc_Min and Loc_Max, If the element compared is less than the minimum value, Loc_Min is set to the index of that element. If the element to be compared is greater than the maximum value, the Loc_Max is set to the index of that element. Processing in this manner, we'll be able to find out the maximum and the minimum element of the current array for a particular pass.
3. Minimum element is interchanged with the first element of the current array, whereas the maximum element is swapped with the Last element of the current array.
4. Now the first and the last element are in proper index position after the first pass.
5. In the next pass,the array starts with the second element and ends at the second last element. The same process is repeated to find the minimum and maximum elements of the current array and they are swapped with the first and the last element of the current array ,i.e., 2^{nd} and 2^{nd} last element of the main array.
6. Now two elements at the beginning and two elements at the end of the main array are now in their proper position. Array gets sorted in ascending order from the beginning and in decreasing order from the end.
7. So after kth pass, k elements at the beginning and k elements at the end of the array are at their proper positions, and thus the length of the array is decreased by two elements after each pass.

8. In the first pass the current length of array is n, in second pass the current length reduces to n-2, then to n-4 and in the last pass the current length will be only 2 in $(n/2)^{th}$ pass.

We find the location of minimum element i.e., Loc_Min and the location of maximum element i.e., Loc_Max in each pass and swap them with the first and last elements of the current array. Now we define various possibilities of the locations of Loc_Min and Loc_Max and the method to put them in their right positions:

1. Minimum and the maximum elements are at proper positions i.e., [Loc_min = i and Loc_Max = (n-1-i)], then we do nothing.
2. Minimum element is found at the maximum index and the maximum element is found at the minimum index. i.e., Loc_Min = n-1-I and Loc_Max= i., then swap the minimum and maximum elements. After swapping, minimum element will be at i^{th} location and maximum element will be at the $(n-1-i)^{th}$ location.
3. Minimum element is found at the maximum index,but the maximum element is found somewhere else except the minimum index. i.e., Loc_Min = n-i-1 but Loc_Max != i, then:
   i) First swap the minimum element of the current array(at Loc_Min) with element at i^{th} location
   ii) Then swap the maximum element of the current array (Loc_Max) with the element at $(n-1-i)^{th}$ location.

   Note: If the reverse is done then we'll not be able to get the minimum element at the i^{th} location.
4. Maximum element is found at the minimum index, but the minimum element is found somewhere else except the maximum index. i.e., Loc_Min != n-1-i but Loc_Max = i, then:
   i) First swap the maximum element of the current array(at Loc_Max) with element at $(n-1-i)^{th}$ location
   ii) Then swap the minimum element of the current array (Loc_Min) with the element at i^{th} location.

   Note: If the reverse is done then we'll not be able to get the minimum element at the i^{th} location.
5. None of the minimum and maximum elements are at their proper position i.e., Loc_Min != i and Loc_Max! = n-1-i, then perform these steps irrespective of the order:
   i) Swap the minimum element of the current array (Loc_Min) with the element at i^{th} location.
   ii) Swap the maximum element of the current array(at Loc_Max) with element at $(n-1-i)^{th}$ location

IV. THE CODE FOR THE UPGRADED SELECTION SORT

```
void Upgraded_Selection_Sort(int Array[],int n)
{
        int i,j,Loc_Min,Loc_Max;


        for(i=0;i<n/2;i++)
        {
                Loc_Min = i;
                Loc_Max = i;

                for(j=i+1;j<n-i;j++)
                {

                        if (Array[j]> Array[Loc_Max])
                        {
                                Loc_Max = j;
                        } /* end if */
                        else if (Array[j]< Array[Loc_Min])
                        {
                                Loc_Min = j;
                        } /* end else if */
                } /* end for */
                if (i == Loc_Max && n-1-i == Loc_Min)
                {
```

```
                Swapping(Array,Loc_Min,Loc_Max);
            } /* end if */
            else
            {
            if ((Loc_Min == n-1-i) && (Loc_Max != i))
                    {
                    Swapping(Array,i,Loc_Min);
                    Swapping(Array,n-1-i,Loc_Max);
                     } /* end if */
            else if ((Loc_Max == i) && (Loc_Min != n-1-i))
                    {
                            Swapping(Array,n-1-i,Loc_Max);
                            Swapping(Array,i,Loc_Min);
                    } /* end else if */
                    else
                    {
                            if(Loc_Min != i)
                            {
                                    Swapping(Array,i,Loc_Min);
                            } /* end if */
                            else if(Loc_Max!= n-1-i)
                            {
                                    Swapping(Array,Loc_Max,n-1-i);
                            } /* end else if */
                    } /* end else */
            } /* end else */
    } /* end for */
} /* end Upgraded_Selection_Sort */


void Swapping(int Array[10],int Temp_Loc_Min,int Temp_Loc_Max)
{
    int temp;

    temp = Array[Temp_Loc_Min];
    Array[Temp_Loc_Min] = Array[Temp_Loc_Max];
    Array[Temp_Loc_Max] =temp;
}/* end Swapping */
```

V. PERFORMANCE EVALUATION:

Analysis of the Upgraded selection sort is straightforward [1]. The first pass makes n-1 comparisons as the array is of length n; the second pass makes n-3 comparisons as now the length of the current array has reduced to n-2, the third pass makes n-5 comparisons as the legth of the current array is n-4 and so on. Therefore, there is a total of

$$(n-1) + (n-3) + (n-5) + \ldots\ldots\ldots\ldots\ldots + 1 = n * n$$

Comparisons, which is $O(n^2)$. In the worst case , when both the minimum and maximum elements are explicitly interchanged with other elements, the number of interchanges is (n-1) in each pass . In the average case ,when the minimum and the maximum elements are placed in their proper position after only one interchange, the number of interchanges is (n/2) , and in best case no interchanges are required. There is little additional storage required (except to hold a few temporary variables). The sort may therefore be categorized as $O(n^2)$. The selection sort can be used for any file for which n is small.

VI. CONCLUSION AND FUTURE WORK

The upgraded Selection sort proposed in this paper , works by repeatedly selecting the minimum or the maximum elements of the key attribute and placing them in their proper position in the list. Thus the array get

sorted from both ends simultaneously. It is sorted from the starting end in ascending order and from the end of the array in descending order. The proposed algorithm sort the file in $O(n^{2})$ time. It requires little additional storage .(except to hold a few temporary variables) . Thus the upgraded selection sort algorithm is an in place sorting algorithm.

We suppose that the comparisons done in order to find the minimum and the maximum element of the current array in each pass may be reduced further. We leave this as our future work.

VII. REFERENCES

[1] Data structure using C and C++ , Yedidyah Langsam,Moshe J. Augenstein, Aaron M. Tenenbaum.