Improved Selection Sort Algorithm

J. B. Hayfron-Acquah, Ph.D.
Department of Computer Science
Kwame Nkrumah University of Science and Technology
Kumasi, Ghana

Obed Appiah
University of Energy and Natural Resources
Sunyani, Ghana

K. Riverson, Ph.D.
CSIR
Accra, Ghana

ABSTRACT

One of the basic problems of Computer Science is sorting a list of items. It refers to the arrangement of numerical or alphabetical or character data in statistical order. Bubble, Insertion, Selection, Merge, and Quick sort are most common ones and they all have different performances based on the size of the list to be sorted. As the size of a list increases, some of the sorting algorithm turns to perform better than others and most cases programmers select algorithms that perform well even as the size of the input data increases. As the size of dataset increases, there is always the chance of duplication or some form of redundancies occurring in the list. For example, list of ages of students on a university campus is likely to have majority of them repeating. A new algorithm is proposed which can perform sorting faster than most sorting algorithms in such cases. The improved selection sort algorithm is a modification of the existing selection sort, but here the number of passes needed to sort the list is not solely based on the size of the list, but the number of distinct values in the dataset. This offers a far better performance as compared with the old selection sort in the case where there are redundancies in the list.

General Terms

Algorithms, Sorting Algorithms

Keywords

Algorithms, sorting algorithms, selection sort, improved selection sort, redundancies in dataset

1. INTRODUCTION

One of the basic problems of Computer Science is *sorting* a list of items. This is the arrangement of a set of items either in increasing or decreasing order. The formal definition of the sorting problem is as follows:

Input: A sequence having n numbers in some random order

$$(a_1, a_2, a_3, \ldots.. a_n)$$

Output: A permutation $(a'_1, a'_2, a'_3, \ldots.. a'_n)$ of the input sequence such that

$$a'_1 \leq a'_2 \leq a'_3 \leq \ldots.. a'_n$$

For instance, if the given input of numbers is 59, 41, 31, 41, 26, 58, then the output sequence returned by a sorting algorithm will be 26, 31, 41, 41, 58, 59 [1].

Sorting is considered as a fundamental operation in Computer Science as it is used as an intermediate step in many programs. For example, the binary search algorithm (one of the fastest search algorithms) requires that data must be sorted before the search could be done accurately at all times. Data is generally sorted to facilitate the process of searching. As a result of its vital or key role in computing, several techniques for sorting have been proposed. The bubble, insertion, selection, merge, heap, and quick sort are some of the common sorting algorithms. Due to high number of sorting algorithms available, the best one for a particular application depends on various factors which were summarised by Jadoon et al. (2011) as:

- The size of the list (number of elements to be sorted).
- The extent up to which the given input sequence is already sorted.
- The probable constraints on the given input values.
- The system architecture on which the sorting operation will be performed.
- The type of storage devices to be used: main memory or disks [4].

Almost all the available sorting algorithms can be categorized into two groups based on their difficulty. The complexity of an algorithm and its relative effectiveness are directly correlated [5]. A standardized notation i.e. Big O(n), is used to describe the complexity of an algorithm. In this notation, the O represents the complexity of the algorithm and n represents the size of the input data values. The two groups of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection sort and O(nlogn) which includes the merge, heap & quick sort.

1.1 Selection Sort Algorithm

The concept of the existing selection sort (SS) algorithm is simple and can easily be implemented as compared to others such as the merge or quick sorting. The algorithm does not need extra memory space in order to perform the sorting. The SS simply partition the list into two main logical parts, the sorted part and the unsorted part. Any iteration picks a value form the unsorted and places it in the sorted list, making the sort partition grow in size while the unsorted partition shrinks for each iteration. When adding to the sorted list, the algorithm makes sure that the value is added at the right position to ensure an order sequence of the sorted partition. The process is terminated when the number of items or the size of the unsorted is one (1). The procedure to select a value to be moved to the sorted list will return minimum value or maximum value in the unsorted partition, which will be swapped to position the item correctly.

1.2 Algorithm: Selection Sort (a[], n)

Here ***a*** is the unsorted input list and **n** is the size of the list or number of items in the list. After completion of the algorithm the array will become sorted. Variable **max** keeps the location of the maximum value in each iteration.

```
k← n-1
Repeat steps 3 to 6 until k=1
Set max=0
```

```
***Repeat for count=1 to k***
        ***If (a[count]>a[max])***
                ***Set max=count***
        ***End if***
 ***Interchange data at location k and max***
***Set k ← k - 1***
```

Table 1.0 shows the time complexity of the algorithm in three different situation of the input list.

Table 1.0: Time Complexity of Selection Sort Algorithm

Best case	Worst case	Average case
O(n^2)	O(n^2)	O(n^2)

Various improved selection sorting algorithms have been proposed and all works better than the Selection Sort Algorithm. Optimized Selection Sort Algorithm (OSSA) starts sorting the array from both ends. In a single iteration, the smallest and largest elements in the unsorted part of the array are searched and swapped[2]. The array is logically partition into three parts; lower-sorted, unsorted, upper-sorted. The search for the maximum and minimum is done in the unsorted partition and the minimum is moved to the lower-sorted and the maximum to the upper-sorted. All values in the upper-sorted are greater or equal to the values in the lower-sorted. The process is continued until the whole list or array is sorted [3]. The algorithm is able to half the run time of the selection sort, *O(n^2)/2*, which is better but still exhibit a time complexity of *O(n^2)*.

The concept of the Enhance Selection Sort Algorithm (ESSA) is to memorize the location of the past maximum and start searching from that point in the subsequent iteration[2]. This enables the algorithm to avoid having to search for the maximum values form the beginning of the unsorted partition to the end. This technique limits the number of comparisons the algorithm performs during each iteration, hence performing better than the existing selection sort algorithm. The arrangement of the elements of the list influences the run time greatly. The same set of data may take different times to be sorted as a result of their arrangement. The average case of the algorithm is however *O(n^2)*.

Hybrid Select Sort Algorithm (HSSA) uses a technique that prevent the algorithm from performing unnecessary iterations by evaluating the content of the unsorted partition for ordered sequence so as to terminate quickly. When the list is fully sorted or partially sorted, its run time is better when compared with the existing selection algorithm. The modified selection sort algorithm uses a single Boolean variable 'FLAG' to signal the termination of execution based on the order of the list, a[i-1] >= a[i] >=a[i+1] [6]. The best scenario is when the list is already ordered, here the algorithm terminate during the first pass, hence will have a run time of *O(n)*. What this means is that, when data is not ordered, the algorithm behaves generally like the old selection sort algorithm.

2. CONCEPT OF IMPROVED SELECTION SORT ALGORITHM

Generally, large data sample will contain a couple of repetitions. For example sorting the ages of citizens of a country of population of about 10 million will contain a lot of repetitions. If age ranges between 0 to 100 then each age value could have a frequency of about 100,000 (10,000,000/100). In terms of population, more than half will be below the ages of fifty (50). The existing selection sort will execute such list in the order of *O(n^2)* in the worst case scenario, but the proposed algorithm can do better. The main concept of the proposed algorithm is to evaluate the data in the list and keep track of distinct values in the list. This makes it possible to perform multiple swapping at each pass unlike the existing selection sort which performs at most 1 at each pass, hence reducing the run time for sorting the list. The technique used is simple; a queue is maintained to keep the locations of all the values that are the same as the value that is held as the Minimum or Maximum. At the end of the list, all the locations on the queue are swapped into their respective positions. Where the subsequence search will begin from can be computed as (i ← i+x), where i points to the start of the unsorted partition and x is the number of items that were dequeue. The worst case happens when there are no repetition in the list, but can guarantee best case run time *O(n)* when all the values in the list are the same or the number of distinct values is relatively small.

2.1 Improved Selection Sort Algorithm

1. Initialise i to 1
2. Repeat steps 3-5 until the i equals n.
3. Search from the beginning of the unsorted part of the list to the end.
4. Enqueue the locations of all values that are the same as the Maximum value.
5. Use the indices on the queue to perform swapping.

Example of Improved Selection Sort (Ascending Order)

List – A[n]

Queue – Q[n]

Initial List

A	2	2	1	5	2	5	4	4	5	5

Q										

1st Pass

A	2	2	1	5	2	5	4	4	5	5

Q	2									

Index 2 added to the queue.

2nd Pass

A	1	2	2	5	2	5	4	4	5	5

Q	1	2	4							

Indices 1, 2 and 4 added to the queue because they all store the same value as the minimum value during the second (2nd) Pass.

3nd Pass

A	1	2	2	2	5	5	4	4	5	5

Q	6	7								

Indices 6 and 7 added to the queue because they all store the same value as the minimum value during the third (3rd) Pass.

4th Pass

A	1	2	2	2	4	4	5	5	5	5

Q	6	7	8	9						

Indices 6, 7, 8 and 9 added to the queue because they all store the same value as the minimum value during the fourth (4th) Pass.

Sorted List

A	1	2	2	2	4	4	5	5	5	5

The list is sorted at the end of the fourth iteration or pass. The existing selection sort will take more time to sort the same list.

Another example with original list as follows

Original List

A=[2, 2, 4, 2, 1, 2, 2, 3, 3, 4, 4, 2, 3, 4, 1, 2, 3, 4, 4, 2]

----

1 Pass

A [2, 2, 4, 2, 1, 2, 2, 3, 3, 4, 4, 2, 3, 4, 1, 2, 3, 4, 4, 2]

Q [4, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Indices 4,14 are added to the queue

After swapping

A [1, 1, 4, 2, 2, 2, 2, 3, 3, 4, 4, 2, 3, 4, 2, 2, 3, 4, 4, 2]

----

2 Pass

A [1, 1, 4, 2, 2, 2, 2, 3, 3, 4, 4, 2, 3, 4, 2, 2, 3, 4, 4, 2]

Q [3, 4, 5, 6, 11, 14, 15, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Indices 3, 4, 5, 6, 11, 14, 15, 19 are added to the queue

After swapping

A [1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 3, 4, 3, 3, 3, 4, 4, 4]

----

3 Pass

A [1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 3, 4, 3, 3, 3, 4, 4, 4]

Q [12, 14, 15, 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Indices 12, 14, 15, 16 are added to the queue

After swapping

A [1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4]

----

4 Pass

A [1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4]

Q [14, 15, 16, 17, 18, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Indices 14, 15, 16, 17, 18, 19 are added to the queue

After swapping

A [1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4]

The Improved Selection Sort Algorithm (ISSA) is content sensitive, in that the nature of data distribution of the list will greatly influence the run time of the algorithm. The run time of the ISSA depends on the number of distinct values that are found in the list to be sorted. If the number of distinct values is big or equal to n, then the run time of the algorithm can be approximated as $O(n^2)$. However, if the number is very small, the algorithm completes the sorting in the order of *O(n).*

Pseudocode

```
A[n]
Queue[n]            // Same size as the size of the array
i ← 0
while i < (n-1)
        Rear ← 0
        Max ← A[i]
        Queue[Rear] ← i
        j←i+1
        while j<(n)
                if Max < A[j]
                        Max ← A[j]
                        Rear ← -1
                If Max = A[j]
                        Rear ← Rear + 1
                        Queue[Rear] = j
         //Perform the swapping of values
        Front ← 0
        While (Front <= Rear)
                Temp ←A[Queue[Front]]
                A[Queue[Front]] ← A[i]
                A[i] ← Temp
                i ← i + 1
                Front ← Front + 1
```

3. ANALYSES OF ISSA

The Improved Selection Sort Algorithm is very simple to analyse, considering the fact that the time complexity or run time of the algorithm depends on two main factors.

1. Size of list (n)
2. Number of distinct values in the list. *dV*

Run Time = O(n.dV)

Table 1shows the runtime of a set of n values with different number of distinct values.

Table 1 Run time of Improved Selection Sort Algorithm (ISSA)

Number of Distinct Values	Run Time	Big-O
1	T = n	O(n)
2	T= 2n	O(n)
3	T=3n	O(n)
…	…	…
n-2	T = (n-2)n	O(n^2)
n	T = n^2	O(n^2)

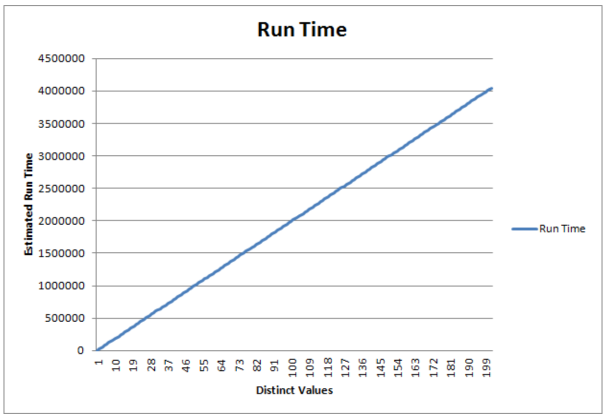

Figure 1: illustrates the relationship between distinct values in the list and the run time of the ISSA.

Fig 1: Run time of list of size n and number of distinct values using ISSA. The performance of the Improved Selection could also be enhanced by introducing the FLAG concept in the HSSA to terminate sorting when the list is already sorted.

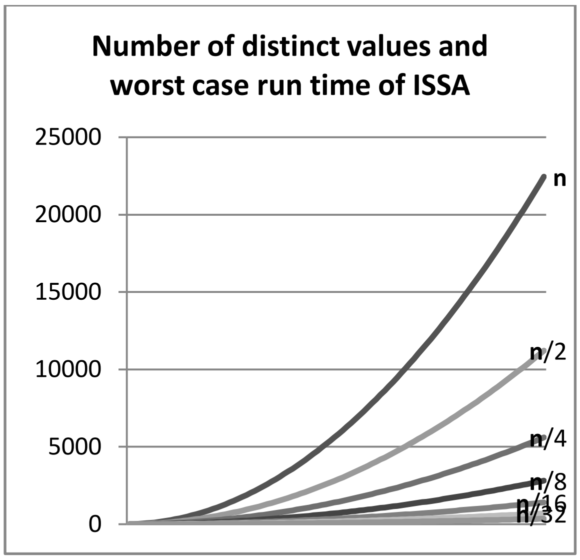

Figure 2: Number of distinct values and run time complexity of ISSA.

Fig 2 illustrates the relationship between the number of distinct values in a list and the time needed to sort it. The number is illustrated as a ratio of the size of the list (n). If the number of distinct value is half the size of the list, then the algorithm will take about half the time the old selection sort algorithm takes. From figure 2, as the number of distinct values decreases, the run time for the sorting also decrease.

Decreasing distinct values:

$$\frac{n}{1},\frac{n}{2},\frac{n}{3},\frac{n}{4},\dots,\frac{n}{n-2},\frac{n}{n-1},1$$

4. ANALYSIS OF SS, OSSA, ESSA, HSSA AND ISSA WITH ASAMPLE DATASET

A given set of data of size 1000 was finally used to analyse the performances of the various selection sort algorithms including Improved Selection Sort Algorithm (ISSA). The number of redundancies in the set was quantified in terms of percentages and 11 different sets of values were used to test the algorithms. The data redundancies in set 1 through 11 were 0%, 10%, 20%, 30%, 40%,50%, 60%, 70%, 80%, 90%, 100%. Table 2 illustrates the run times for the various algorithms on the various categories of the dataset.

Table 2: Estimated run times of various Selection Sort algorithms when input dataset were not sorted

Red.	SS	OSSA	ESSA	HSSA	ISSA
'0%	499500	250000	371580	498501	499500
'10%	499500	250000	377050	498501	449550
'20%	499500	250000	375967	498501	399600
'30%	499500	250000	378348	498501	349650
'40%	499500	250000	383873	498501	299700
'50%	499500	250000	398155	498498	249750
'60%	499500	250000	399608	498501	199800
'70%	499500	250000	418296	498500	149850
'80%	499500	250000	433374	498495	99900
'90%	499500	250000	463134	498465	49950
'100%	499500	250000	49950	998	1000

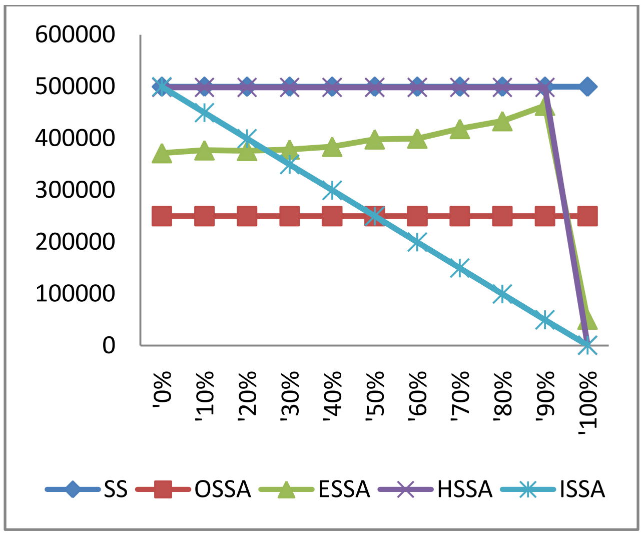

Figure 3: Estimated run times of various Selection Sort algorithms when input dataset were not sorted

The actual comparison of these selection sort algorithms could be done when the dataset is randomized. Here the performance of the improved selection sort algorithm (ISSA) recorded best performance when the percentages of redundancies exceeds 50%

5. CONCLUSION

This paper proposed a new selection sort algorithm which performs better than the existing selection sort algorithm and in most cases may have a run time in order of *O(n)* which is ideal for sorting relatively large set of data. The strength of the algorithm depends on the distinct values in the list and where there are more of such redundancies or repetitions in the list, it performs better than the existing selection sort algorithm and also a couple of the optimized selection sort. In situation where the number of distinct values is very small, the algorithm may perform better than even the quick sort and merge sort algorithm which have run time *O(nlogn).*

6. REFERENCES

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. Introduction to Algorithms. MIT Press. Cambridge. MA. 2nd edition. 2001

[2] Jadoon, S., Solehria, S. F., Qayum, M., “Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study” International Journal of Electrical & Computer Sciences IJECS-IJENS Vol: 11 No: 02, 2011

[3] Jadoon, S., Faiz S., Rehman S., Jan H., “Design & Analysis of Optimized Selection Sort Algorithm”, IJEC-IJENS Volume 11 Issue 01, 2011.

[4] Khairullah, M. “Enhancing Worst Sorting Algorithms”. International Journal of Advanced Science and Technology Vol. 56, July, 2013

[5] Kapur, E., Kumar, P. and Gupta, S., “Proposal of a two way sorting algorithm and performance comparison with existing algorithms”. International Journal of Computer Science, Engineering and Applications (IJCSEA) Vol.2, No.3, June 2012

[6] “Design and Analysis of Hybrid Selection Sort Algorithm”. International Journal of Applied Research and Studies (iJARS) ISSN: 2278-9480 Volume 2, Issue 7 (July- 2013) www.ijars.in