Visualization of Searching and Sorting Algorithms

Bremananth R, Radhika.V and Thenmozhi.S

Abstract—Sequences of execution of algorithms in an interactive manner using multimedia tools are employed in this paper. It helps to realize the concept of fundamentals of algorithms such as searching and sorting method in a simple manner. Visualization gains more attention than theoretical study and it is an easy way of learning process. We propose methods for finding runtime sequence of each algorithm in an interactive way and aims to overcome the drawbacks of the existing character systems. System illustrates each and every step clearly using text and animation. Comparisons of its time complexity have been carried out and results show that our approach provides better perceptive of algorithms.

Keywords—Algorithms, Searching, Sorting, Visualization.

I. INTRODUCTION

Visualization of algorithms’ sequence is an important process to learn various hidden steps, which are involved dynamically. The advantages of visualizing algorithms are: Easy to learn with different set data, Understand hidden steps of algorithms, Memory usages and Time management strategy. The first well-known visualization presented by Baecker, it was in videotape format. It shows the animation of nine different sorting algorithms. This videotape allows students to watch the behavior of the algorithm rather than try to imagine its actions from a verbal explanation or from several static images [1]. Brown Algorithm Simulator and Animator (B(ALS)A) is a major interactive algorithm animation system developed at Brown University [2]. In this system Students were able to control the animation by starting, stopping and replaying the animations. A later version of this system is BALSA-II added color and some rudimentary sounds. Brown University created another algorithm animation system. It does not erase and redraw each image as the previous animation systems did. It is able to produce smoother more cartoon-like animations. A later version of this system is XTANGO[3]. A New Interactive Modeler for Animations in Lectures (ANIMAL) is a newer visualization system incorporating lessons learned from pedagogical research Developed at the University of Stegen in Germany [4]. In this paper, some of searching and sorting algorithms are explained visually. Interaction with this tool can be achieved through the exploration of existing default visualizations, through the direct manipulation of graphics objects. This tool is designed for three different groups of users such as students, instructors and software developers. This will be very interactive which means the user verifies the algorithms by different set of data. We explain the following algorithms in this paper.

Section 2 describes visualization of searching algorithms and Sorting algorithms have been discussed in section 3. Section 4 deals result and analysis of both kinds of algorithms. Finally, concluding remarks and future enhancements are described in section 5.

II. VISUALIZING SEARCHING TECHNICS

Interaction with our system can be achieved through the exploration of existing default visualizations, through the direct manipulation of graphical objects. This will provide the way by selection of concepts (Searching or Sorting) which we want and also select the algorithm based on the selected concept. The inputs for the selected algorithm are obtained from the user. Visualization process starts by clicking the start button. The Buttons Pause and Resume are used to suspend and resume the process of visualization. In both Searching and sorting algorithms, an appropriate message is displayed for each process. In Searching algorithms, when process starts, the component (labels) that contains the element to be searched moves through the list based on the selected algorithm until a match is found. In sorting algorithms, when process starts, the positions of the components (labels) that contain the elements to be sorted are interchanged. This process continues until all the elements are sorted. It contains the following algorithms.

A. Sequential Search

In searching algorithms, the user has to give how many number of input, the set of data and a number to be searched. Then select the particular algorithm from the list and then the...
visualization of the selected algorithm is shown with the given inputs. A Sequential search is one of a search method, also known as linear search that is suitable for searching a list of data with a particular value. It operates by checking every element of a list one at a time in a sequential order until a match is found.

**Algorithm** Sequential Search (List, Target, N)

- **List** - the elements to be searched
- **Target** - the value being searched for
- **N** - the number of elements in the list
- **pos** - the value of the position from where repaint starts

**Step 1:** calls ct.d() //function that starts the timer

**Step 2:** For i = 1 to N do

**Step 3:** if (Target = List[i]) then

**Step 4:** Stop the program

**Step 5:** End if

**Step 6:** pos = i //stores the position

**Step 7:** call repaint();

// It moves searching element through out the list
// until a match is found

**Step 8:** End for

**Step 9:** calls ct.d1() //function that stops the timer

Fig.1 illustrates choosing the algorithm from the given list. Timer is started with the algorithm using the function ct.d(). The element to be searched is called the Target. For elements 1 to n the Target is compared with the elements in the List starting from the first element. In each step the index value of the element in the List with which the Target is compared is stored in variable pos.

Then repaint() function is called to move the element to the next position of the List as depicted in Fig.2.

**B. Binary Search**

In binary search, we first compare the target with the element in the middle position of the array. If there's a match, we can return immediately. If the target is less than the middle element, then the target must lie in the lower half of the array; if the target is greater than the middle element then target must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

**Algorithm** Binary Search (List, Target, N)

- **List** - the elements to be searched
- **Target** - the value being searched for
- **N** - the number of elements in the list
- **pos** - the value of the position from where repaint starts

thread_var -variable to control the movement of the labels to be exchanged
Step 1: calls ct.d() // function that starts the timer  
Step 2: start = 1  
Step 3: end = N  
Step 4: while start <= end do  
Step 5: middle = (start + end)/2  
Step 6: if (List[middle] < Target) then  
Step 7: start = middle + 1  
Step 8: pos = start  
Step 9: While (thread_var < 3) do  
Step 10: call repaint()  
// It moves searching element through specified list  
// until a match is found  
Step 11: End While  
Step 12: End if  
Step 13: if (List[middle] > Target) then  
Step 14: end = middle - 1;  
Step 15: pos = end;  
Step 16: call repaint()  
// It moves searching element through specified list  
// until a match is found  
Step 17: End if  
Step 18: if (List[middle] = Target) then  
Step 19: Stop the program  
Step 20: End if  
Step 21: End while  
Step 22: calls ct.d1() // function that stops the timer

The Binary search option is chosen from the given list as in Fig.4.

Fig. 4 Binary search- Choosing the algorithm

The program starts and consecutively the timer is also started using the function ct.d(). The value 1 is stored in variable start and N in variable end. It finds the middle value using the formula middle = (start + end)/2. It compares the middle element with the Target. When the Target is greater than the middle element, it starts searching the upper half by assigning start = middle + 1 and storing the start value in variable pos and repaint() function is called to move the Target element to the first position of the upper half as shown in Fig.5.

Fig. 5 Binary search searching the upper half

It applies the above steps for the upper half until the Target is found. It stops when Target is found as illustrated in Fig.6. When the Target is less than the middle element, the program starts searching in the lower half by assigning end = middle - 1 and end value is stored in variable pos and repaint() function is called to move the Target element to the lower half. It applies the above steps for the lower half until the Target is found.

When Target is equal to the middle element the middle value is stored in variable pos and repaint() function is called to place the Target in middle. Then the program stops and the timer is stopped using the function ct.d1(). It searches only in a sorted list. When the list is unsorted, it sorts the list first and then starts searching.

Fig. 6 Binary search- Element found

C. Interpolation Search

Interpolation search is an algorithm for searching a given target value in an indexed array that has been sorted in ascending order. In each search step it calculates where in the remaining search space the target might be, based on the values at the bounds of the search space and the value of the target, usually via a linear interpolation. The value actually found at this estimated position is then compared to the target value. If it is not equal, then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position.

Algorithm Interpolation Search (Target, N)
**N** - the number of elements in the list

**pos** - the position of the first element to be exchanged

**pos1** - the position of the second element to be exchanged

**flg** - temporary variable

**Target** - the value to be searched

**thread_var** - Variable to control the movement of the labels to be exchanged

**Step1**: calls ct.d()//function that Starts the timer

**Step2**: calls Interpolation (list, Target)

**Step3**: calls ct.d1 ()//function that stops the clock

Interpolation (sortedArray, Target) // Returns index of //Target in sortedArray, or -1 if not found

**Step1**: low = 0;

**Step2**: high = sortedArray.length - 1;

**Step3**: while (sortedArray [low] < Target & & sortedArray [high] >= Target) do

**Step4**: Mid = low + ((Target - sortedArray [low]) * (high - low)) / (sortedArray [high] - sortedArray [low])

**Step5**: pos = mid

**Step6**: calls repaint ()

**Step7**: if (sortedArray [mid] < Target) then

**Step8**: low = mid + 1;

**Step9**: pos = mid

**Step10**: calls repaint ()

**Step11**: else if (sortedArray [mid] > Target) then

**Step12**: high = mid - 1;

**Step13**: pos = mid

**Step14**: calls repaint ()

**Step15**: else

**Step16**: pos = mid

**Step17**: calls repaint ()

**Step18**: End While

**Step19**: if (flg=0) then

**Step20**: if (SortedArray [low] = Target) then

**Step21**: pos = low

**Step22**: calls repaint ()

**Step23**: Else

**Step24**: pos = 0

**Step25**: calls repaint ()

**Step26**: End if

First the Interpolation search option is chosen from the given list as in Fig.7. The timer is started using the function ct.d(). When the list is unsorted, it sorts the list first and then starts searching.

**Target** is the value to be searched. The value 0 is assigned to the variable low and sortedArray.length -1 is assigned to variable high. When the sortedArray’s first element is less than the Target and the last element of the sortedArray is greater than the Target, mid value is calculated using the formula mid= low + ((Target - sortedArray [low]) * (high - low)) / (sortedArray [high] - sortedArray [low]) and the mid value is stored in variable pos and the repaint() function is called to move the Target element to the position stored in variable pos.

When middle value is less than Target assign low=mid+1 and mid value is assigned to variable pos and repaint() function is called to move the element. When middle element of sortedArray is greater than Target then assign high=mid-1 and variable pos is assigned the mid value and repaint() function is called to move the element. When both the above conditions are not satisfied, variable pos is assigned the mid value and repaint() is called to move the element and a variable flg is set to 1. When the flg value is 0 and sortedArray[low] is equal to Target the low is assigned to variable pos and repaint() function is called as depicted in Fig.8. Otherwise variable pos is assigned the value 0 and then repaint() function is called to swap the element. Then the program stops and the timer is stopped using the function ct.d1().

First the Interpolation search option is chosen from the given list as in Fig.7. The timer is started using the function ct.d(). When the list is unsorted, it sorts the list first and then starts searching.

![Fig. 7 Interpolated search- Choosing the algorithm](image-url)

![Fig. 8 Interpolated search-Element found](image-url)
### III. SORTING ALGORITHM

In sorting algorithms the user has to give how many number of inputs and the set of data. Then select the particular algorithm from the list and then the visualization of the selected algorithm is shown with the given inputs.

#### A. Selection sort

The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for remaining of the list (to the consecutive positions)

Effectively, we divide the list into two parts: the sub list of items already sorted, which we build up from left to right and is found at the beginning, and the sub list of items remaining to be sorted, occupying the remainder of the array.

**Algorithm** Selection Sort (List, N)

- *List* - the elements to be put in order
- *N* - the number of elements in the list
- *pos* - the position of the first element to be exchanged
- *pos1* - the position of the second element to be exchanged
- *t* - temporary variable
- *min* - variable to store the Minimum value
- *thread_var* - variable to control the movement of the labels to be exchanged

**Step 1:** calls ct.d() //Starts the timer

**Step 2:** For i=0 to N-1 do

**Step 3:** min = i

**Step 4:** For j = i+1 to N do

**Step 5:** if (List[j] < List[min]) then

**Step 6:** min = j;

**Step 7:** End if

**Step 8:** End for

**Step 9:** pos = i

**Step 10:** pos1 = min

**Step 11:** if (min != i)

**Step 12:** while (thread_var < 3)

**Step 13:** Calls repaint() //It Exchanges the elements

**Step 14:** End While

**Step 15:** End If

**Step 16:** t = List[i]

**Step 17:** List[i] = List[min]

**Step 18:** List[ min ] = t

**Step 19:** thread_var = 0

**Step 20:** End For

**Step 21:** calls ct.d1()//function that stops the timer

The algorithm starts by selecting the selection sort option from the list as in Fig.9 and the timer is started using the function ct.d().

![Selection Sort- Choosing the algorithm](image1)

**Fig. 9 Selection sort- Choosing the algorithm**

From the first element to the last element, find the smallest element and its position is assigned to the variable *min*. Variable *pos* is assigned the value *i* that represent the position in the array starting from 1 and variable *pos1* is assigned the value *min*. Then repaint() function is called to swap the values in the positions given in variable *pos* and *pos1* as illustrated in Fig.10.

![Selection Sort- Performing swapping](image2)

**Fig.10 Selection sort- Performing swapping**

The process continues until the *List* is sorted. The sorted *List* is shown in Fig.11 and the timer is stopped using the function ct.d1().
B. Bubble sort

The bubble sort algorithm makes number of passes through the list of elements. On each pass it compares adjacent element values. If they are out of order, they are swapped. We start each of the passes at the beginning of the list. On first pass, once the algorithm reaches the largest element, it will be swapped with all of the remaining elements, moving it to the end of the list. The second pass will move the second largest element down the list until it is in the second to last location. The process continues with each additional pass moving one more of the larger values down in the list. If on any pass there are no swaps, all of the elements are now in order and the algorithm can stop.

Algorithm Bubble Sort (List, N)

List -the elements to be put in order
N -the number of elements in the list
pos -the position of the first element to be exchanged
pos1 -the position of the second element to be exchanged
t -temporary variable
thread_var -Variable to control the movement of the labels to be exchanged

Step1: calls ct.d() //function to start the timer
Step2: For i=N-1 to 0 Step -1 do
Step3: For j=0 to i do
Step4: if (List[j] > List[j+1]) then
Step5: pos=j
Step6: pos1=j+1
Step7: while (thread_var<3)
Step8: Calls repaint () //It Exchanges the elements
Step9: End While

Fig.12 Bubble sort- Choosing the algorithm

The bubble sort algorithm makes N-1 number of passes through the List of elements where N is the number of input values. On each pass it compares adjacent element value. If they are out of order their position are stored in variables pos and pos1 and repaint() function is called to swap the elements in the two positions as in Fig.13.

Fig.13 Bubble sort- Performing swapping

The program stops after the N-1 number of passes and the List is sorted as given in Fig.14. The timer is stopped using the function ct.d1().
C. Shell Sort

Shell sort was developed by Donald L. Shell. It begins by dividing the full list of values as a set of interleaved sub lists. On the first pass, it divides the list into two sub lists. On the second pass, it could be further divided into sub lists. The size of the set to be sorted gets smaller with each pass through the list, until the sub list’s length become 1. As the size of the set decreases, the number of sets to be sorted increases.

**Algorithm Shell Sort (List, N)**

- **List** - the elements to be put in order
- **N** - the number of elements in the list
- **pos** - the position of the first element to be exchanged
- **pos1** - the position of the second element to be exchanged
- **h** - variable to divide the list
- **v** - variable to store the list element, to compare
- **thread_var** - variable to control the movement of the labels to be exchanged

**Step1:** calls ct.d()//function that starts the timer

**Step2:** h=len

**Step3:** do

**Step4:** h=h/2

**Step5:** for i=h to n-1 do

**Step6:** v=List[i]

**Step7:** j=i

**Step8:** while ((j>=h) & (list [j-h]>v)) do

**Step9:** pos=j

**Step10:** pos1=j-h

**Step11:** while (thread_var<3) do

**Step12:** Calls repaint ()//It Exchanges the elements

**Step13:** End While

**Step14:** List[j]=List [j-h]

**Step15:** j=j-h

**Step16:** End While

**Step17:** List[j]=v

**Step18:** thread_var=0

**Step19:** End For

**Step20:** End while (h>1)

**Step21:** calls ct.d1()//function that stops the timer

The shell sort algorithm begins after choosing the shell sort option from the list and clicking the start button. Timer is started using the function ct.d(). The List is divided into two sub list. After dividing the List into two the first element of both the sub list are compared. If they are out of order their index value in the array is stored in variables pos and pos1 respectively and the repaint() function is called to swap the element as depicted in Fig.15.

Similarly the consecutive elements are compared. The sub list are further sub divided and above procedure is repeated. The sub lists are sub divided until the length of the sub lists become one. By repeating the above procedure the whole list is sorted. The program stops after displaying the sorted list as given in Fig.16. The timer is stopped by calling the function ct.d().

![Fig.14 Bubble sort- Sorting process completed](image1)

![Fig.15 Shell sort- Performing swapping](image2)

![Fig.16 Shell sort- Sorting Completed](image3)
IV. RESULT ANALYSIS

A. Analysis of Searching Algorithms

Table I depicts the comparison of searching algorithms with average values. In searching algorithms, number of inputs are 8. Its average runtime is given in seconds. From analyzing Fig. 17, we came to know that both the interpolation and binary search would be time consuming rather than sequential search. This is due to sorting the data before searching the elements. But basically an interpolation search will be less time consuming, secondly binary search will be less time consuming one while entering a sorted list as input to these algorithms. Sequential search does not need a sorted list for searching.

<table>
<thead>
<tr>
<th>No. of Inputs</th>
<th>Sequential Searching in Sec.</th>
<th>Binary Searching in Sec.</th>
<th>Interpolation Searching in Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
<td>3.1</td>
<td>3.8</td>
</tr>
<tr>
<td>3</td>
<td>2.6</td>
<td>7.8</td>
<td>7.7</td>
</tr>
<tr>
<td>4</td>
<td>3.2</td>
<td>23.2</td>
<td>23.7</td>
</tr>
<tr>
<td>5</td>
<td>4.3</td>
<td>27</td>
<td>28.1</td>
</tr>
<tr>
<td>6</td>
<td>6.4</td>
<td>32.3</td>
<td>34.2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>51.2</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>9.6</td>
<td>60.8</td>
<td>61.4</td>
</tr>
</tbody>
</table>

Fig. 17 Comparison of searching algorithms-with average values

B. Analysis of Sorting Algorithms

Table II illustrates the comparison between the sorting algorithms with average values. Sorting algorithms are analyzed with 8 input values. Its average runtime is given in seconds. From analyzing Fig. 18, we came to know that Selection sort is less time consuming when compared to Bubble and shell sort. Among the three sorting algorithms Bubble sort will be the most time consuming algorithm. On bubble sort each time adjacent elements are compared and swapped when needed. This process will be repeated from the beginning of the array until all the elements are sorted. Therefore Bubble sort will be the most time consuming algorithm.

<table>
<thead>
<tr>
<th>No. of Inputs</th>
<th>Selection Sorting in Sec.</th>
<th>Bubble Sorting in Sec.</th>
<th>Shell Sorting in Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2.8</td>
<td>2.8</td>
<td>2.8</td>
</tr>
<tr>
<td>3</td>
<td>5.8</td>
<td>6.7</td>
<td>6.7</td>
</tr>
<tr>
<td>4</td>
<td>7.6</td>
<td>13.4</td>
<td>9.6</td>
</tr>
<tr>
<td>5</td>
<td>9.6</td>
<td>25</td>
<td>13.4</td>
</tr>
<tr>
<td>6</td>
<td>16.3</td>
<td>35.5</td>
<td>20.1</td>
</tr>
<tr>
<td>7</td>
<td>16.8</td>
<td>40.3</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>17.3</td>
<td>55.7</td>
<td>30.8</td>
</tr>
</tbody>
</table>

Fig. 18 Comparison of searching algorithms-with average values
V. CONCLUSION AND FUTURE ENHANCEMENT

This system is implemented for visualizing some of the searching and sorting algorithms. This is a helpful tool for all kinds of learners/scholars to easily understand the implicit sequences of algorithm. Here the users are allowed to select the options, either searching or sorting. Then they are allowed to give input and they can select the algorithms from the list and the algorithm is explained visually.

In future to enhance and continue this project, the system may include more algorithms for searching and sorting. Visualization can also be done for other kinds of algorithms. Voice can further be included to the system, to give more interaction for the end users.

REFERENCES


Bremananth R received the B.Sc and M.Sc. degrees in Computer Science from Madurai kamaraj and Bharathidhsan University, India in 1991 and 1993, respectively. He has obtained M.Phil. degree in Computer Science & Engineering from Bharathiar University. He has received his Ph.D. degree in the Department of CSE, PSG College of Technology, India, Anna University, Chennai. Department of Computer Applications, Sri Ramakrishna Engineering College, Coimbatore, India. He has 16 years of teaching experience and published several research papers in the National and International Journals and Conferences. He has received M N Saha Memorial award for the year 2006 by IETE. His fields of research are pattern recognition, computer vision, image processing, biometrics, multimedia and soft computing. Dr. Bremananth is a member of Indian society of technical education, advanced computing society, ACS and IETE.

Radhika.V has completed her B.Sc. in Computer Technology in Sri Ramakrishna Engineering College, affiliated to Anna University Coimbatore, Tamil Nadu, India. She is also a research student with Dept. of Computer Applications Sri Ramakrishna Engineering College, affiliated to Anna University Coimbatore, Tamil Nadu, India.

Thenmozhi.S has completed her B.Sc. in Computer Technology in Sri Ramakrishna Engineering College, affiliated to Anna University Coimbatore, Tamil Nadu, India. She is placed as system associate in iGate Global Solutions, Bangalore, India. She is also a research student with Dept. of Computer Applications Sri Ramakrishna Engineering College, affiliated to Anna University Coimbatore, Tamil Nadu, India.