TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
PULCHOWK, LALITPUR

A report on DSA project:

VIZUO

Data Manipulation in Action

DATE: 2079-12-03

SUBMITTED BY:

AAYUSHA ODARI (077BCT006)
KASHISH BATAJU (077BCT035)
KRISHALA PRAJAPATI (077BCT038)
MAMATA MAHARJAN (077BCT043)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
PULCHOWK CAMPUS, IOE,TU

Acknowledgements

We want to offer our appreciation to everyone who helped this project be completed successfully. Before anything else, we want to thank Mrs. Bibha Sthapit, our professor, for all of her help and support this semester. Her insightful teaching patterns enabled us to raise the quality of our work. We also want to express our gratitude to the Pulchowk Campus' Department of Electronics and Computer Engineering for providing us with the tools and space we needed to conduct our project. We would like to extend our heartfelt appreciation to the open-source community as well as the creators of the many algorithms used in this project. The completion of this project would not have been feasible without their assistance.

Last but not least, we'd like to express our gratitude to one another for all of the effort, commitment, and cooperation that went into this project. It has been a pleasure working as a team, and we have all learned a lot from one another.

We would like to thank you all once more for your assistance with this effort.

Abstract

VIZUO is a web application with interactive animations that showcase how sorting algorithms, searching algorithms, and depth first search algorithms work. These visualizations allow you to better understand how these algorithms operate and how they differ from one another.

The visualization tool is implemented using JavaScript and will enable users to visualize various sorting and searching algorithms, including bubble sort, quick sort, binary search, and linear search. It will also incorporate DFS-based algorithms for maze generation, such as recursive backtracking, which will allow users to generate and visualize complex mazes. The project will focus on providing a user-friendly interface that will enable users to interact with the visualization tool easily.

Overall, the DSA visualization tool will help students and developers better understand how sorting, searching, and maze generation algorithms work in practice. It will serve as a useful educational tool for those learning DSA concepts and as a reference tool for developers implementing these algorithms in their projects.

Table of Contents

Objectives

- To implement our theoretical knowledge in DSA to develop a useful program.
- To develop a web-based visualizer for various algorithms to help users understand them better.
- To learn about sorting algorithms, searching algorithms in a better way.
- To be able to work as a team in upcoming major projects
- To showcase various sorting algorithms such as Bubble Sort, Merge Sort, Quick Sort, and Heap Sort, and provide a visual representation of how they work.
- To demonstrate different search algorithms such as Linear Search, Binary Search, and Interpolation Search, and explain their performance characteristics.
- To implement a maze generation algorithm using depth-first search and explain how it works.

Introduction

The aim of this project is to develop a web-based visualizer for sorting algorithms, searching algorithms, and maze generation algorithm using depth first search. These visualizations allow you to better understand how these algorithms operate by animating the changes made to the input array at each step of the sorting algorithm.

This project focuses on three main components: sorting algorithms, searching algorithms, and maze generation algorithms. Sorting Algorithms are used to sort a data structure according to a specific data relationship. Sorting can be done based on some key references like numbers, alphabets. Searching is a process of retrieving a specific item in a collection of data. Our searching algorithm visualizer demonstrates the operation of two different search algorithms: linear search and binary search on a given list of data. Similarly, our web visualizer also includes a maze generation feature. Maze generation using depth first search is a common algorithmic approach to creating randomized mazes. One of the benefits of using depth first search for maze generation is that it results in mazes with a single path from start to finish. This makes it ideal for generating mazes for games or puzzles that require a clear solution path

Overall, this project aims to create a platform that simplifies the learning and understanding of sorting algorithms, searching algorithms, and maze generation algorithms. In the following sections, we will discuss the design and implementation of our project in more detail.

Implementation

The project was implemented using HTML, CSS, and JavaScript. The front-end of the tool was designed using HTML and CSS, while the algorithms were implemented using JavaScript. HTML and CSS was used to create a bar chart representation of the array, with the height of each bar representing its value. We also implemented sliders that allows the user to control the speed of the animation and size of the array to be sorted in the sorting algorithm.

1. Sorting Algorithms

a. Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Bubble sort has a time complexity of O(n^2), which makes it inefficient for large lists.

PSEUDOCODE for Bubble Sort:

```
bubbleSort(array, n) {
for i = 0 to n-1
  for j = to n-i
      if (array[j] > array[j + 1])
      {
         swap array[j] and array[j+1]
      }
}
```

The function takes an input array and sorts it using the bubble sort algorithm. The outer loop iterates from the first element to the second last element of the array, and the inner loop iterates from the first element to the second last element minus the current outer loop index. The if condition inside the inner loop checks if the current element is greater than the next element, and if it is, it swaps the two elements.

The visualization of the sorting process was implemented by showing a bar graph of the input array. The height of each bar represents the value of the corresponding element in the input array. The visualization shows the changes made to the input array at each step of the sorting algorithm.

b. Selection Sort

The selection sort algorithm works by repeatedly finding the minimum element in the unsorted part of the array and swapping it with the first element in the unsorted part.

PSEUDOCODE for Selection sort

```
selectionSort(array, n) {
for i = 0 to n-1
    let minIndex = i;
    for j = i+1 to n-1
      if (array[j] < array[minIndex]) {
        minIndex = j;
      }
    next j
    if (i !== minIndex) {
       swap array[i] and array[minIndex]
    }
  next i
}
```

We used a nested loop to implement the algorithm, where the outer loop iterates over the unsorted part of the array, and the inner loop finds the minimum element in the unsorted part.

c. Insertion Sort

Insertion sort is a sorting algorithm that builds a sorted list one element at a time. It works by taking one element from the unsorted portion of the list and inserting it into its proper place in the sorted portion of the list. This process is repeated until the list is sorted. Insertion sort has a time complexity of O(n^2) in the worst case, but it can perform better on partially sorted lists.

PSEUDOCODE for insertion sort

```
procedure insertionSort(A: list of sortable items)
  n = length(A)
  for i = 1 to n - 1 do
     j = i
     while j > 0 and A[j-1] > A[j] do
        swap(A[j], A[j-1])
        j = j - 1
     end while
  end for
end procedure
```

d. Quick Sort

Quick sort is also a divide-and-conquer algorithm that works by selecting a pivot element, partitioning the list around the pivot, and then recursively sorting the sublists on either side of the pivot. Quick sort has a time complexity of O(nlogn) in the average case, but it can have a worst-case time complexity of O(n^2) if the pivot is poorly chosen.

PSEUDOCODE for quick sort

```
quickSort(arr[], beg, end) {
   if (beg < end) {
      pivot = partition(arr, beg, end);
      quickSort(arr, beg, pivot – 1);  // Before pi
      quickSort(arr, pivot + 1, end); // After pi
   }
}



/* This function takes first element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than pivot) to
left of pivot and all greater elements to right of pivot */
partition (arr[], beg, end)
{
   left=beg
   right=end
   pivot=beg
   if (beg < end)
   {
        key = arr[pivot]
        while (left < right)
        {
            while (arr[left] <= key && left < end)
                left++
            while (arr[right] > key)
                right--
            if (left < right)
                swap(arr[left], arr[right])
        }
        swap(arr[pivot], arr[right])
        return right
    }
    else
 return
}
```

e. Merge Sort

Merge sort is a divide-and-conquer algorithm that works by dividing the list into smaller sublists, sorting them recursively, and then merging the sublists back together. Merge sort has a time complexity of O(nlogn) and is considered one of the most efficient sorting algorithms for large data sets.

PSEUDOCODE for merge sort

```
Merge_sort(array) {
    if (array_size< 2)
          return array
    end if
    middle = (array_size / 2)
    left_array= array.slice(0, middle)
    right _array= array.slice(middle)
    Merge_sort(left_array)
    Merge_sort(right_array)

     i = 0
     j = 0
     k = 0

    loop while (i < left_array_size && j < right_length_array)
           if (left_array[i] < right_array[j])
                     array[k] = left_array[i]
                      i++
            end if
            else
                   array[k] = right_array[j]
                   j++
            end else
         k++
      end of while loop

    loop while (i < left_array_size)
                    array[k] = left_array[i]
                      i++
                      k++
      end of while loop

      loop  while (j < right_array_size)
                     array[k] = right_array[j]
                      j++
                      k++
     end of while loop
     }
```

f. Shell Sort

Shell sort is an optimization of insertion sort that sorts elements far apart from each other and then gradually reduces the gap between them until the entire list is sorted. It works by dividing the list into smaller sublists, sorting them with insertion sort, and then merging the sublists back together. Shell sort has a time complexity that depends on the gap sequence used, but it can be faster than insertion sort for larger data sets.

PSEUDOCODE for shell sort

```
Shell_sort(array) {
   Gap=array_size
   loop while (Gap>1)
        Gap = ((Gap+1) / 2);
        for  j = Gap to  j <array_size
               for  i = (j - Gap) to  i >= 0
                      k=i+Gap
                      if (array[k] < array[i])
                             swap (array[k] , array[i])
               i=i-Gap
               end for loop of i
        j++
        end for loop of j
   end of while loop
}
```

g. Heap Sort

Heap sort algorithm:

1. Build a max-heap from the input array or list. This is done by repeatedly swapping elements to satisfy the heap property, which states that the parent node should be greater than or equal to its children.

2. Extract the maximum element from the heap (which is always the root node of the heap) and add it to the sorted output array. This step reduces the size of the heap by one.

3. Restore the heap property by percolating down the new root element. This is done by swapping the root element with its larger child until the heap property is satisfied.

4. Repeat steps 2 and 3 until the heap is empty

Pseudo-Code:

```
    Where A:is an array of unsorted numbers and i : rootNode
1.   Subroutine: Heapify(length(A),i)
     Largest = i;
     leftChild = 2*i+1
     rightChile = 2*i+2

     if(leftChild<Length(A) && A[leftChild]>A[largest])
      Then largest = leftChild

     if(rightChild<Length(A) && A[rightChild]>A[largest])
      Then largest = rightChild

     If largest is not the root
     Then temp = A[largest]
          A[largest] = A[i]
          A[i] = temp
          Recursive call on heapify(length(A), largest)
     END Heapify
2.   HeapSort(length(A))
     //Build heap tree (rearrange array)
     For i = length/2 - 1 until  i >=0 ; i--
          Call heapify(length,i)
     END LOOP
     //One by one extracting the element from heap
     For i = n-1 until i > 0 ; i - -
        Swap the position of
            A[i] and A[0]     //move current root to the end

        Call Heapify(i, 0) // on the reduced heap and node 0 as root node
      END FOR
```

h. Radix Sort

Algorithm for radix sort:

1. Determine the maximum number of digits in the input list, which is the number of times the sorting process should be repeated. This is also the maximum number of digits among all the input numbers.

2. For each digit position, starting from the least significant digit and moving to the most significant digit:
   a. Create 10 buckets (0 to 9) and initialize them as empty.

   b. For each input number, extract the digit at the current position and place the number into the corresponding bucket based on that digit.

   c. Concatenate the buckets in order, starting from 0 and going up to 9, to form a new list of numbers.

   The resulting list of numbers is sorted

Pseudo-Code:
Arr[]: unsorted array of numbers and n = length(arr) and exp: position of digit.

```
1. Subroutine: getMax() a utility function to get the maximum value in
   arr[]
       Let mx = arr[0]
         for(i = 0 to i<n)
             if(arr[i] > mx)
                  Mx = arr[i]
             End if
         End loop
   Return mx;

2. Subroutine: countSort(arr,n,exp) //a function to do counting sort of
   arr[] according to the digit represented by exp
   Output = new array(n)
   Count = new array(10)

   For i =0 to i < n
       Count[i] = 0

   //store count of occurrence in count[]
   For i = 0 to i < n
        count[int(arr[i]/exp)%10]++

   //change the count[i] so that count[i] now contains actual position of
   this digit in output[]
```

```
For i = 1 to i < 10
        Count[i] += count[i-1]

//build the output array
For i = n-1 to i >= 0 ; i - -
         output[int[count((arr[i]/exp) %10)] -1] = arr[i]
         count[int(arr[i]/exp)%10]--
END LOOP

//assign the new array to arr for recalculations
For i=0 to i<n
  Arr[i] = output[i]
  END loop

END Subroutine

3. RadixSort(arr,n)
   Let m = getmax(arr,n)

   For exp = 1; int(m.exp) >0 ; exp *=10
      countSort(arr, n, exp)
   END radixSort()
```

2. Searching Algorithms

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Here are two main search algorithms: linear search and binary search.

a. Linear Search

Linear search, also known as sequential search, is a simple search algorithm that checks every element of the dataset until it finds the desired value. This algorithm works best on small datasets and unordered lists. The time complexity of linear search is O(n), where n is the number of elements in the dataset. This means that the time taken to search the dataset increases linearly as the number of elements increases.

PSEUDOCODE for Linear Search:

```
Linear_search(array) {
      pos=-1
  for i=0 to array_size
      value=array[i]
     if(value==num)
           pos=i
           display (Element found)
            break
     end if
   next i
```

```
 if(pos=-1)
        display(Element not found)
  end If
}
```

b. Binary Search

Binary search is a more efficient algorithm used to search a sorted dataset. This algorithm divides the dataset in half repeatedly until the desired value is found. It works by checking the middle element of the dataset and comparing it to the value being searched for. If the value is less than the middle element, the algorithm continues searching the left half of the dataset. If the value is greater than the middle element, the algorithm searches the right half of the dataset. This process continues until the value is found or the dataset is exhausted.

The time complexity of binary search is O(log n), where n is the number of elements in the dataset. This means that the time taken to search the dataset increases logarithmically as the number of elements increases. Binary search is faster than linear search for large datasets, especially when the dataset is sorted.

PSEUDOCODE for Binary Search

```
Binary_search(sorted_array){
   start=0
   end=(sorted_array_size-1)
   pos=-1
   loop while(start<=end)
       midd=((start+end)/2)
       value=sorted_array[midd];
       if(value=num)
           pos=midd
           display (Element found)
           break
       end if
      if(value>num)
          end=midd-1
      end if
      else
          start=midd+1
      end else
   end while loop
 if(pos=-1)
      display(Element not found)
 end if
}
```

In conclusion, choosing the appropriate search algorithm depends on the dataset and the specific requirements of the program. Linear search is simple and easy to implement, but it can be slow for large datasets. Binary search, on the other hand, is more efficient for sorted datasets, but requires the dataset to be sorted before searching.

3. Maze Generator

Maze generation using depth first search is a common algorithmic approach to creating randomized mazes. The algorithm begins by selecting a starting cell, marking it as visited, and adding it to a stack. From there, it continues by selecting a neighboring cell that has not yet been visited and marking it as visited, pushing it onto the stack, and repeating the process until all possible moves have been exhausted.

When there are no more unvisited cells adjacent to the current cell, the algorithm backtracks to the previous cell in the stack and continues the process from there, selecting the next unvisited neighboring cell and adding it to the stack. This continues until all cells in the maze have been visited, resulting in a complete maze with no loops or isolated cells.

One of the benefits of using depth first search for maze generation is that it results in mazes with a single path from start to finish. This makes it ideal for generating mazes for games or puzzles that require a clear solution path.

Algorithm for maze generation using depth first search algorithm

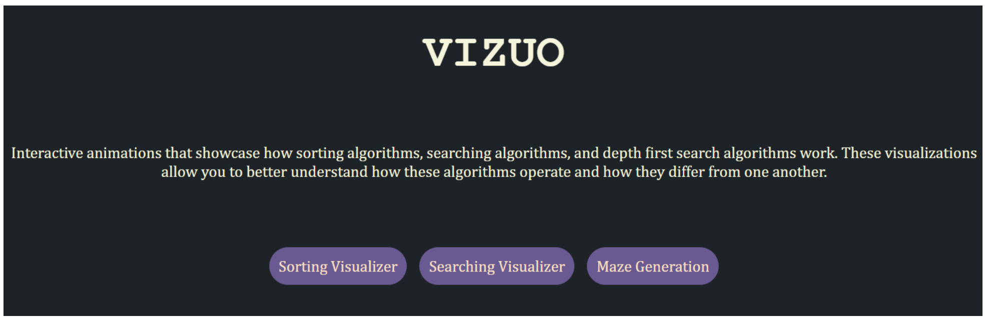
1. Randomly select a node (or cell) N.
2. Push the node N onto a stack S.(N-current cell)
3. Mark the cell N as visited.
4. Randomly select an adjacent cell A(random neighbour) of node N that has not been visited. If all the neighbours of N have been visited:
    - Continue to pop items off the stack S until a node is encountered with at least one non-visited neighbour - assign this node to N and go to step 4 (recursive backtracking)
    - If no nodes exist: stop.
5. Break the wall between N and A.
6. Assign the value A to N.(currentcell = random unvisited neighbour)

Go to step 2.

Results

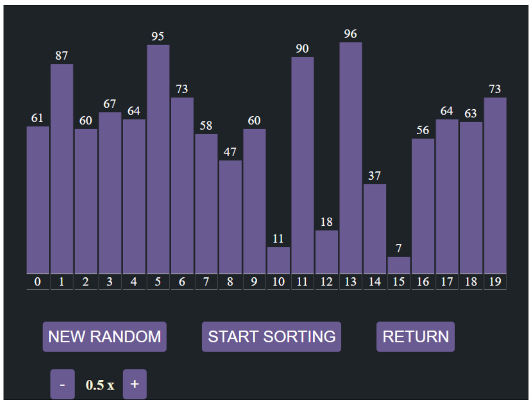
The main aim of this project was to enhance our ideas about algorithms implementation and graphs for maze and to visualize them in a webpage. The project was built using modern web technologies such as HTML, CSS, and JavaScript. We were successful in animating different sorting and searching algorithms and created a maze using depth first algorithm.
For the sorting algorithm, we implemented eight different sort algorithms. The visualizer allowed us to see how each algorithm sorted a set of random data in real-time.
Similarly, for the searching algorithm, the visualizer can perform linear search and binary search. Data is searched in unsorted data in linear search whereas the random data is sorted first using Shell sort and then binary search is implemented.
For the maze generator, we implemented a depth first search algorithm to generate a maze in real-time. The user is able to select the size of the maze of different levels and the algorithm would generate a maze that could be solved by the user.

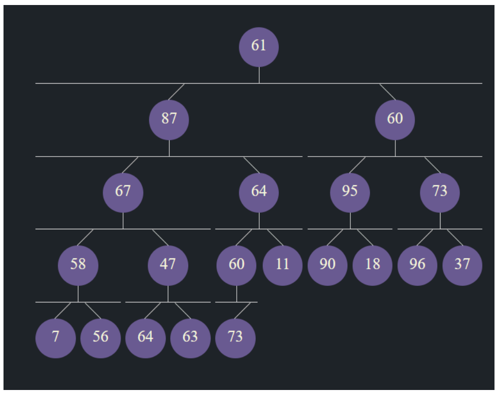
Some snippets of our project VIZUO

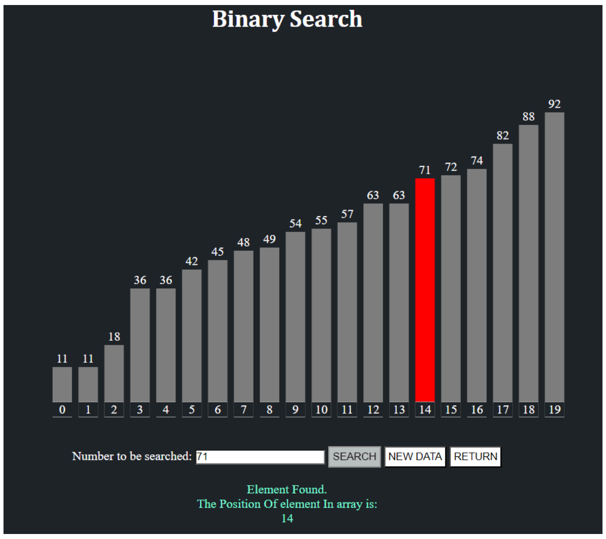


VIZUO

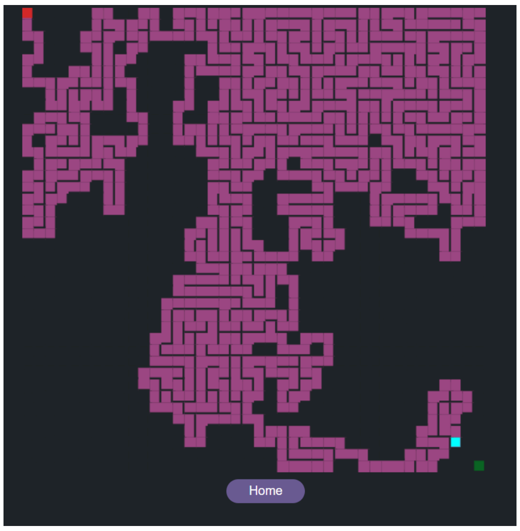
Interactive animations that showcase how sorting algorithms, searching algorithms, and depth first search algorithms work. These visualizations allow you to better understand how these algorithms operate and how they differ from one another.







Element Found.
The Position Of element In array is:
14



Problems faced and solutions

We for sure faced a few hurdles during the project span and were successful in addressing them in a clear and concise manner. Developing a project to visualize searching, sorting, and maze generation algorithms using DSA can be a challenging task. The creation of effective algorithms, visualizations of how the data is actually being altered, looking for accurate resources, debugging sessions, performance, and user experience were initially difficult for us. One must have a good understanding of algorithms, data structures, programming languages, and user interface design. By carefully planning and executing the project, we overcome these challenges and now deliver a high-quality product that effectively visualizes the algorithms.

To fight back against the challenges of DSA visualization, we used solutions such as proper research, collaboration, testing, optimization, and group debugging sessions. We designed efficient algorithms, created effective visualizations, optimized the performance, prioritized the user experience, and ensured compatibility by testing the project on different browsers, devices, and operating systems. Our approach to this project proved to be good enough to deliver a high-quality product that effectively visualizes the algorithms, although there are still changes to be made.

Limitations and future enhancements

As this was our first attempt at making a web with all the functionalities of html, css and javascript, saving the progress of the web for the future, and so on, we were not able to achieve all these functionalities in our first attempt. There are still many features that we can add to VIZUO to make it a better version than it is now. In this version, we were not able to make the web page independent of the size of the device at the same time as per what we thought, but we still tried to arrange the orientation of pages that best fits the four of our devices based on the knowledge we had about css. We could only develop the front end part of the project where the backend remains untouched. We have not added the feature for the user to enter the unsorted arrays themselves.

In short, there are a lot of things that we could add to this version VIZUO in the future. We are planning to add some extra levels of visualizing like using trees and boxes that we imagine in our mind, as it is mediocre at this stage. So in the future, we will add multiple algorithms or visualization of data structures construction and backend features as well. And we will also make it independent of device size for a better experience. We would also add the option of choosing the way of visualizing (like trees or bars or boxes etc.) so that the project would be more interesting. Overall, we plan on making VIZUO really advanced with the addition of new features.

Conclusion and Recommendations

During the execution of this project, we practiced a lot of searching, sorting, and maze generation algorithms using DSA. This project not only assisted us in developing a surface knowledge of the Algorithms, but it also assisted us in developing a surface knowledge of the web development with html, css and javascript. We learned how to work with different elements of frontend development, and understand different algorithms in depth. After the completion of the project, the best thing we achieved was the skill and the mindset to work as a team and get an opportunity to teach and learn at the same time.

For those with the intention of web development for visualizers, we recommend giving a fair amount of time to grasping the concepts of the proper and most efficient use of the functions available in javascript. The concepts of html, css and javascripts are indeed a must when it comes to web development. It is recommended that beginners understand the entire process of web development without skipping a bit and feel free to search for solutions to the tiniest problems online.

References

https://www.geeksforgeeks.org/
https://medium.com/swlh/solving-mazes-with-depth-first-search-e315771317ae
https://en.wikipedia.org/wiki/Maze_generation_algorithm