Principles of Programming Languages (CSE-526)

ALGORITHM VISUALIZER

PROJECT REPORT

Team:

1. **Kushaal Hulsoor (114776851)**
2. **Priyanka Bhosale (114779627)**

Table of Contents

Problem Description 3

Evaluation and Comparison of Existing Works 3

Tasks and subtasks 4

- Environment setup and API definitions 4
- Basic Layout design & UI components integration 4
- Array and Graph visualization operations implementation 4

Tools and project setup 4

UI and Component layout 5

About React 6

API definitions 6

- Array API calls: 6
- Graph Nodes API calls : 7

Flow and design 8

Driver Programs and Examples 8

Implementation details 9

Testing and Evaluation 12

Credits 14

Part I:
Problem and plan

1. Problem Description

Algorithm Visualizer, as the name suggests, allows users to visualize different algorithms using various animations, which would help them gain a clearer understanding of the logic.

Currently, there are multiple tools ([4]) available wherein the user can select any of the listed algorithms and get a visual representation of them with ease. However, these tools are very specific and focus on a particular algorithm, rather than providing a generic way where users can build their logic. For example, if a user wants to add his algorithm, it would require that the user understand the source code of the visualizer project and add his algorithm as a feature. This is time-consuming and tedious.

These problems can be avoided if the tool allows users to write their algorithms and visualize them. This tool can also be used by beginners who are keen to learn data structures/ algorithms to create their code, style it, and visualize the algorithm in their customized format.

The solution is to build an interactive web application that would allow users to dynamically visualize the data structure operations using Javascript (ReactJS). The user would be provided with a set of API functions that can be used to build his logic. These functions would be a very basic set of operations that can be applied to a data structure, calling them would manipulate the data structure in the memory, and also trigger a visual representation of the data change.

2. Evaluation and Comparison of Existing Works

Following are a few existing tools that support data structure visualization:

First one ([1]), is a tool that supports a huge bunch of existing algorithms (eg. Stacks, sorting, etc) and visualizes their operations using different input data. The only control the user has is over what data a specific algorithm would work on, and display the operations in animations. This is strictly restricting users to visualize from a predefined set of existing data structures. The tool is written mainly in ActionScript3 which is hard to maintain and inefficient due to its tightly coupled architecture. Using ReactJs would be a perfect solution for this use case to increase the performance.

The second one ([2]), makes it best not only for learners to get started with program flow and execution but also for experts who would want to debug their programs by tracking the variable values at every state. However, this visualization is complex to understand especially if the user is a beginner and the user has zero control over the styling of the visual elements.

3. Tasks and subtasks

This project can be planned and categorized into 3 major tasks:

a. Environment setup and API definitions

The React environment will be set up with all the required Components and method definitions required for the project. The basic program structure is created with executable boilerplate code and helper functions that are needed. We are planning to use the javascript in-built function eval() to execute the code input.

b. Basic Layout design & UI components integration

This involves creating and integrating all the required components with the template codes such that a fully functional UI is rendered. In this stage, the app must allow any input from the user and should be able to execute it to provide an output. Everything would work except the integration between the user code and the visualization.

c. Array and Graph visualization operations implementation

This step involves adding visualizations to all the atomic operations that are possible on an Array and Graph data structure. The idea is to allow users with a generic set of options rather than confining the tool to specific data structures. Hence we would add every possible operation to the array elements or graph nodes/edges in a way that allows the user to implement any data structure flow that he intends to, including stacks and queues, trees, and graphs.

4. Tools and project setup

In React, any change of state would trigger and re-render the target component view without affecting any other components of the tool. This particular feature of React would be of utmost importance for this tool as we can integrate the code editor component with the visualizer by just changing the state of components rather than worrying about how these changes would be passed between components. In comparison with the existing works technologies; i.e Actionscript3 and HTML, this tool will be more powerful and easily maintainable.

Languages/ Frameworks : Javascript, ReactJs, HTML, CSS
IDEs :Jetbrains Webstorm, Visual Studio Code.

The tentative schedule of weekly tasks to be performed is as follows:

Duration	**Tasks**	**Roles division**
21st - 27th March 2022	Basic Layout & UI components design	Priyanka
28th March - 3rd April 2022	Environment setup and API definitions	Kushaal
4th April - 10th April 2022	Array visualization operations implementation	Priyanka
11th April - 17th April 2022	Graph visualization operations implementation	Kushaal
18th April- 24th April 2022	Styling integration with components	Kushaal, Priyanka
25th April - 1st May 2022	Testing, Report and Presentation completion	Kushaal, Priyanka

Part II:
Design

1. UI and Component layout

We will be splitting the webpage vertically into two individual sections, the left side with the output of visualization, and the right side with a code editor where the user can write his algorithm The user would be able to enter his code on the right division and use the toolbar to either run his code entirely or line by line. He could also reset to the initial state. By choosing the “styling” tab, the same division would allow the user to write his styling declarations which would change the look of the elements.
The help tab would be a static box with quick documentation required to use this tool

2. About React

ReactJs will be the main library that we will be focusing on for this project. Its architecture is in such a way that an entire system can be divided into multiple components with a separate state bound to each component. It has a strong blend of javascript and HTML, which will help us animate user interactive elements easily.

3. API definitions

a. Array API calls:

1. function createArray();
   //It creates and returns an empty instance of an array.

2. function pushFront(element):
   //it adds an element to the beginning of the array and triggers visualization for it.

3. function popFront():
   //it removes an element from the beginning of the array and triggers visualization for it.

4. function pushEnd(element):
   //it adds an element to the end of the array and triggers visualization for it.

5. function popEnd():
   //it removes an element from the end of the array and triggers visualization for it.

6. function popIndex(index):
   //it removes an element from the specified index and triggers visualization for it.

7. function insertIndex(index,element):
   //it adds an element to the specified index of the array and triggers visualization for it.

8. function swapIndices(index1, index2):
   //it swaps the elements from the two specified indices of the array and triggers visualization for it.

9. function leftShift(val):
   //shifts all the elements to the left “val” number of times and triggers visualization for it.

10. function rightShift(val):
    //shifts all the elements to the right “val” number of times circularly and triggers visualization for it.

b. Graph Nodes API calls :

1. function createNode(val):
   //creates the node and triggers visualization for it.

2. function addChild(val, name=null):
   //creates the child node and triggers visualization for it.

3. function removeChild(name=null):
   //removes the edges and the child nodes and triggers visualization for it.

4. function addEdge(n2):
   //adds edge between current node and n2 and triggers visualization for it.

5. function removeEdge(n2):
   //removes the edge from the current node and n2 and triggers visualization for it.

6. function swapNodes(n1,n2):
   //replaces node values and triggers visualization for it.

7. function removeAllEdges(n1):
   //removes all edges (inward and outward) from n1 and triggers visualization for it.

8. function getChild(name=null):
   //returns the child node with given name

4. Flow and design

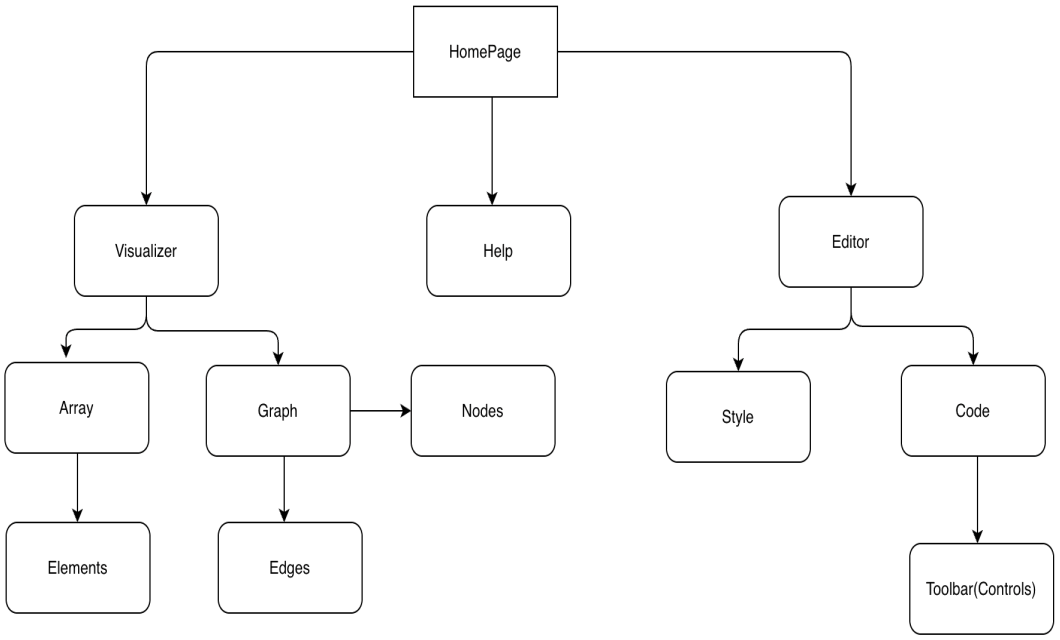


Fig 1. Component Relationship diagram

The above figure represents the relationship between various components that are to be implemented. More details about each component is available in the implementaion section.

5. Driver Programs and Examples

Below are a few examples which the users can write:

1. Creating a circular linked list

```
let start = Graph.createNode(1);
let tmp = start;
for(let i = 2; i <  5; i++){
        next = tmp.addChild(i);
}
Graph.addEdge(next, start);
```

2. Creating a simple BST.

```
let root = Graph.createNode(5);
root.setChildLimit(2);
```

```
let leftCh = root.addChild(2, 'left');
let rightCh = root.addChild(7, 'right');
right.addChild(8, 'right');
left.addChild(3, 'right');
```

3. Inserting a node to above Binary Search Tree

```
let val = 6
while(curr.getChild('left') && curr.getChild('right')) {
        if(val > curr.getVal())
                curr = curr.getChild('right');
        else
                curr = curr.getChild('left');
}
if(val > curr.getVal())
        curr.addChild(val, 'right');
else
        curr.addChild(val, 'left')
```

Part III:
Implementation

Implementation details

The source code can be found on: Source Code (https://github.com/unicomputing/s22-algovis). Since the repository is private, only people with authorized permissions can access it.

As discussed in the design, and adding to it, we have used the following tools:

- The ReactJs framework[3] for creating a single page application and dynamically re-rendering components at runtime which is triggered by user events.
- Javascript DOM interface to handle visual animations.
- HTML SVG tags for rendering the shapes like a rectangle in case of array or circles and arrow in case of trees/ graphs[11].
- CSS for styling.
- Ace web code editor extension [10]

The code size of important functionalities viz. Array.js and graph.js have 181 and 344 LOCs respectively. The code gets compiled in approximately 140-600 ms which is great for a web application.

Let us discuss the implementation in detail:

There are multiple files within this project root that are auto-generated when we install React and are required by that framework. We are going to focus only on files that we have changed or created.

Entire Project is divided into two directories:

1. src
2. tests

1. src

This contains "App.js" file which is the entry point of our project, contains template that renders the navigation bar and its controls and triggers the other components of the page.
All the other components are categorized and implemented in separate directories under "component".

a. editor

Contains "editor.js" that initializes and renders the code editor in the UI. We are using "ace" module which is open-source and provides excellent design and UI to edit code.

b. visualizer

- Visualizer.js

This file contains template code for the visualizer component. It also contains a function "processProgram()" which is the core part of our project. It takes the code user provided as input, initializes a few global constants that are required by the program to run, and the passes it to the eval() function which executes the program in that custom context. This also makes a few changes to the state variables of the component to block user controls when the animations are running, and display errors wherever required.

This also has a method called "consumeQueueAction()". This is triggered whenever the Next button is clicked in Debug mode. All the animations that are to be displayed are stored into a global queue instead of immediately reflecting on the page. Everytime the user clicks the next button, one atomic action is dequeued and executed.

- common.js

These contain classes that are required to be accessed globally, both in the application context and the user program context. The Styles, DebugQueue and Timer classes are declared here.

DebugQueue is the global queue which stores all the animation actions that are to be consumed later.

Timer class stores the current relative timestamp from the instant when the visualization is triggered, which would allow us to execute each animation with constant interval. This allows us to manipulate the speed of the animation.

Styles is used to store all the styling attributes the user provides, and it is applied to all the visualizations. Currently, we support a few attributes like color, edge color, etc. that are configurable. However, it is fairly simple to extend this feature with as many attributes as needed by minimum possible changes.

- Graph.js

This contains the GraphNode class and its functions that manage the graph and trigger the visualization. This includes all the API methods that were discussed in the above sections.
There is a delayed() methos within this class which plays a crucial role in timing all the animations sequentially.

- Array.js

This contains the Array class and its functions that control the array and its functions. This includes all the API methods that were discussed in the above sections. This file follows a similar pattern to that of Graph, with an obvious change of using array elements instead of graphs.

c. Home.js

This file contains template and logic regarding the switching between Styles and Code editor.

2. tests

This contains multiple files each testing a specific algorithm and testing its visualization. To run the tests, we must first run the application and execute each test within the editor of the webpage.

Challenges faced

One major challenge we faced during implementation was to synchronize the time delay between every animation by maintaining consistent order of execution of operations. Although we initially thought of putting the application to sleep between wait intervals, it turned out to be much more complex than we expected since javascript doesn't allow the entire application to pause before executing the next instruction. To resolve this, we used a global timer variable which schedules every animation as a callback function after a constant time. We make sure all the user controls are blocked during this phase to prevent the user from misusing the delayed operations.

Another challenge was to set up the debug functionality. We wanted to allow users to control the animations stepwise. We used a queue to store all the callback functions once the user enters into debug mode. There the user can use the next button which would trigger the callbacks from the queue in "First in first out" order.

There was also a problem with the decision of placement of the next node or element that is created. There were nodes overlapping when many nodes were created, hence blocking the view of few nodes. So

we have implemented a way to allow dragging the nodes using the mouse, allowing users to position the nodes as they intended.

Limitations and Future scope

Although we aimed at building a robust application which would allow users to visualize algorithms just by plugging in their favourite piece of code and making minimum changes to it, there is a never ending list of different operations that are possible to be animated for every data structure. Hence we made sure to make the code as modular as possible so that it takes minimum changes to introduce a new operation on a data structure. The application can easily be incremented to support more data structures and operations without disturbing the existing code.

Setting up and running the application

The application uses node, hence it is required to have node package manager installed in our system.

Then,
The application can be setup by running the below command in project root:
npm install …. installs necessary packages for the project.

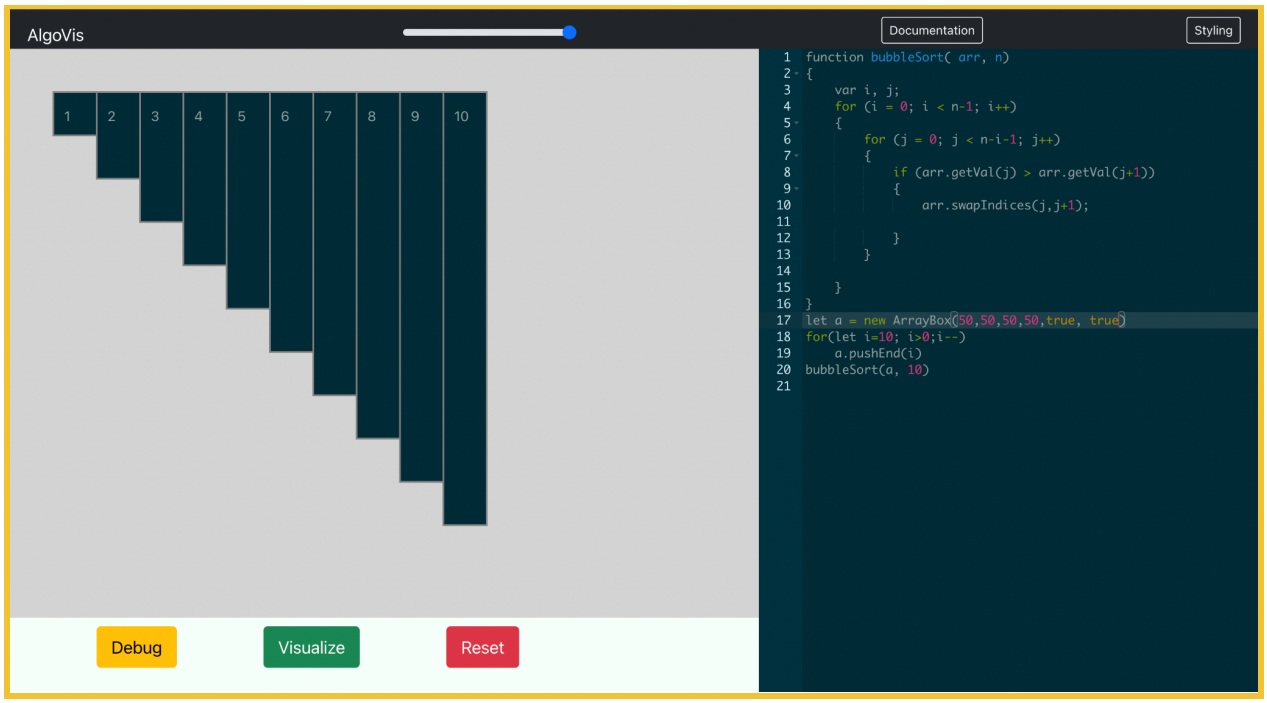
The application can be started by running below command:
npm start …. Compiles the application and hosts it on a web browser.

The application can be run in any browser (tested on Chrome) using address localhost:3000

Part IV:
Testing and Evaluation

Testing and Evaluation

We have tested the application with multiple existing algorithms [8] like Sorting algorithm and Binary Search Trees. As mentioned in the implementation section, we have included all the test scripts within the "tests" directory of our project. The following snapshot is an example of a bubble sort test case:



The test files, each, contain code to visualize a specific algorithm. We can copy the code from the file into the code editor of the application, and visualize it for that algorithm by clicking on the “Visualize” button.

The slider on the top allows the user to control the animation speed.

The Debug button allows the user to enter Debug mode, where one can execute the visualization step-wise.

The editor is the part where the user can write his code by using predefined functions on each data structure that is provided by the project. The documentation[9] allows the user to go through the API specification.

As we planned, this application now allows the user to visualize operations on these data structures for any logic flow. Unike the state of the art[1][4] that we discussed which allowed users to only visualize a specific set of algorithms, users would have more freedom to implement an algorithm of their choice by using simple functions.

Part V:
References

Credits

1. https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
   https://www.cs.usfca.edu/~galles/visualization/source.html
   Copyright 2011 David Galles, University of San Francisco, accessed on Feb 18th, 2022
2. https://pythontutor.com/visualize.html#mode=display
   Philip Guo on January 2010, accessed on Feb 18th, 2022
3. https://reactjs.org/docs/getting-started.html
   Copyright © 2022 Meta Platforms, Inc, accessed on Mar 1st, 2022
4. https://algorithm-visualizer.org/
   https://github.com/algorithm-visualizer
   https://github.com/algorithm-visualizer/algorithm-visualizer/blob/master/LICENSE
   Copyright (c) 2019 Jinseo Jason Park, accessed on Feb 18th, 2022
5. https://visualgo.net/en
   Project Leader & Advisor (Jul 2011-present):
   Dr. Steven Halim, Senior Lecturer, School of Computing (SoC), National University of Singapore (NUS)
   Dr. Felix Halim, Senior Software Engineer, Google (Mountain View).
   accessed on Feb 18th, 2022
6. https://cyberzhg.github.io/toolbox/nfa2dfa
   https://github.com/CyberZHG/toolbox
   Copyright (C) 2007 Free Software Foundation, Inc., accessed on Mar 1st, 2022
7. https://github.com/unicomputing/s22-algovis Source code
8. https://www.geeksforgeeks.org/
   Algorithms for testing
9. https://docs.google.com/document/d/1qfv-EPGBOCkneQRbURvS_p_iKA8lfKGQ5YEMYM4YP_Q/edit Documentation
10. https://github.com/ajaxorg/ace Open source code editor extension for react. Accessed on April 20, 2022.
11. https://www.w3schools.com/graphics/svg_intro.asp SVG open source tutorial.

Contributions:

1. Environment setup and API definitions, Graph visualization operations implementation, Styling integration with components, Testing by **Kushaal Hulsoor**
2. Basic Layout & UI components design, Array visualization operations implementation, Styling integration with components, Testing by **Priyanka Bhosale**