iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.79822

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

Code and Algorithm Visualizer

Divya Dharsni L M[1], Fathima Zuhra A B[2], Mrs. A. Lavanya[3]
[1, 2]*Bachelor of Engineering in Computer Science and Engineering, Adhiyamaan College Of Engineering, (An Autonomous Institution), DR. M. G. R.Nagar, Hosur*
[3]*Assistant Professor, Adhiyamaan College of Engineering, (An Autonomous Institution), Dr. M.G.R. Nagar, Hosur*

Abstract:** In the evolving landscape of computer science, mastering abstract algorithmic concepts remains a significant challenge for learners due to their logical complexity and limited visual representation. The Code and Algorithm Visualizer is an interactive platform designed to simplify these concepts by converting algorithm execution into clear, real-time visual experiences. Developed using the MERN Stack and enhanced with TypeScript, the system ensures scalability, performance, and robust development. The platform allows users not only to observe predefined algorithm simulations but also to actively construct and manipulate data structures in real time. Core concepts such as sorting algorithms, searching techniques, graph traversals, and shortest path computations are visually represented, enabling users to understand how data evolves at each step of execution. A custom-built interactive engine supports dynamic node creation, weighted edge connections, and intuitive drag-and-drop interactions, providing a hands-on learning experience. To further enhance comprehension, the system incorporates step-by-step execution tracking along with real-time explanatory feedback, helping learners understand the reasoning behind each operation. Additional controls such as pause, replay, and speed adjustment allow users to explore algorithms at their own pace, while zoom and pan functionalities ensure clarity even for complex structures. The platform also integrates gamified learning elements, including quizzes, progress tracking, and streak monitoring, to encourage consistent engagement and reinforce learning outcomes. Its responsive and modular design ensures seamless accessibility across devices and supports future scalability with the addition of more algorithms and features. Ultimately, the Code and Algorithm Visualizer bridges the gap between theoretical knowledge and practical understanding by transforming abstract logic into intuitive visual representations. It serves as a powerful educational tool that promotes active learning, strengthens problem-solving abilities, and builds a solid foundation in algorithmic thinking.

I. INTRODUCTION

A. Overview

The Code and Algorithm Visualizer is an advanced platform designed to help computer science students and beginners better understand algorithms through interactive visualization. It allows users to write, execute, and observe algorithms step-by-step using dynamic animations, making it easier to connect theoretical concepts with practical understanding. The system is built using the MERN Stack along with TypeScript, ensuring a scalable and reliable development environment.

One of the main difficulties in learning algorithms is their abstract nature. Many learners struggle to visualize what actually happens during execution. This platform addresses that issue by providing real-time visual feedback, allowing users to clearly see operations such as comparisons, swaps, node visits, and edge relaxations. It supports key topics including sorting, searching, recursion, and graph algorithms, helping users understand how data changes step by step.

A major feature of the system is its interactive simulation engine, which allows users to build and manipulate data structures directly. For example, in graph-based algorithms, users can create nodes, connect weighted edges, and experiment with different scenarios, including more complex cases like negative weights.

The platform also includes automatic algorithm detection, which recognizes code patterns and generates suitable visualizations along with explanations and complexity details. In addition, users can control execution using features like pause, replay, and speed adjustment, making it easier to learn at their own pace.

To keep users motivated, the system includes quizzes, progress tracking, and streak-based learning. The overall design is responsive and modular, allowing it to run smoothly across devices and making it easy to expand in the future.

In summary, the Code and Algorithm Visualizer turns abstract algorithmic concepts into clear and interactive experiences, helping learners build confidence and improve their problem-solving skills.

©IJRASET: All Rights are Reserved

B. Objectives

The main objectives of this project are:

1) To enhance conceptual understanding of algorithms by providing an interactive visualization platform that allows users to manipulate data and observe real-time, step-by-step animations, making abstract computational logic easier to understand.
2) To bridge the gap between theoretical knowledge and practical execution by visually representing internal operations such as comparisons, swaps, edge relaxations, and recursive calls, helping learners build strong mental models of algorithm behaviour.
3) To create a modern, user-friendly, and responsive learning environment with intuitive interfaces, interactive controls, and smooth navigation features such as zooming and panning, supporting both self-paced learning and classroom use.
4) To integrate intelligent assistance through features like automatic algorithm detection, contextual explanations, and a voice-based guidance system, enabling the platform to act as a virtual mentor for learners.
5) To enable hands-on experimentation through an interactive graph engine that allows users to create nodes, define weighted edges (including negative weights), and analyse how structural changes affect algorithm performance.
6) To incorporate gamification elements such as quizzes, progress tracking, execution path visualization, and learning streaks, encouraging consistent engagement and mastery of concepts.
7) To ensure scalability and adaptability by designing a modular architecture using **TypeScript**, allowing easy integration of new algorithms and features.
8) To promote continuous and effective learning by encouraging exploration, experimentation, and a feedback-driven approach that helps learners develop strong problem-solving and algorithmic thinking skills.

II. LITERATURE SURVEY

The literature survey reviews recent advancements in algorithmic pedagogy, full-stack educational frameworks, and interactive visualization technologies designed to bridge the gap between theoretical computation and practical software engineering.

1) Ibrahim and L. Chen, “Generative AI and Real-Time Visualization in CS Education,” IEEE Transactions on Learning Technologies, 2026.

This paper explores the integration of real-time logic generation with dynamic visual feedback. It emphasizes that synchronized auditory and visual cues significantly reduce the cognitive load required to understand non-linear algorithms.

2) R. K. Sharma, “Full-Stack Architectures for Interactive Educational Tools,” Journal of Web Engineering, 2026.

The author discusses the advantages of using the MERN stack for educational platforms, highlighting that high-speed data handling in Node.js and the reactive nature of React are critical for smooth algorithm animations.

3) J. Chen and M. Zhao, “AI-Enhanced Algorithm Visualization for Adaptive Learning,” IEEE Transactions on Learning Technologies, 2026.

This study presents an AI-driven visualization platform that adapts animations based on user performance, improving personalized learning and engagement.

4) R. Kumar et al., “Real-Time Code Visualization Systems for Programming Education,” ACM Transactions on Computing Education, 2026.

The authors developed a system that synchronizes code execution with live visualization, significantly improving student understanding of runtime behaviour.

5) L. Tao et al., “Strict Typing and Scalability in Algorithmic Platforms,” Springer Informatics, 2025.

The authors argue that using TypeScript in educational visualization projects ensures modularity and error-free execution of recursive logic, facilitating the addition of complex data structures.

6) S. K. Mehta et al., “Dynamic Manipulation of Negative Edge Weights in Graph Theory,” Elsevier Procedia Computer Science, 2025.

This paper highlights the educational value of visualizing algorithms that handle negative weights, noting that real-time interaction helps students grasp why certain algorithms like Dijkstra fail where Bellman-Ford succeeds.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

7) Rodrigo Mourato, “State-Based Animations in Modern Web Frameworks,” IEEE Access, 2025.
The study presents a framework for capturing execution snapshots and transforming them into interactive timelines, allowing learners to traverse algorithm steps forwards and backwards.

8) T. Naps et al., “Reflection Activities and Post-Visualization Quizzes,” ACM SIGCSE Bulletin, 2024.
This study reaffirmed that integrating quizzes immediately after an algorithm simulation reinforces the conceptual understanding of the logic just witnessed.

9) L. Wang and S. Patel, “Interactive Graph-Based Learning Platforms for Algorithm Education,” Elsevier Computers & Education, 2025.
This research highlights the effectiveness of graph-based interaction systems in improving comprehension of complex algorithms.

10) R. Pang and X. Lu, “AI-Driven Algorithm Visualization Tool for Python Learning,” IEEE Transactions on Learning Technologies, 2025.
The tool converts Python code into dynamic visuals, helping learners track variable changes and algorithm flow in real time.

11) Vinueza-Morales et al., “Visualization-Based Programming Education: A Bibliometric Study,” Springer Education Informatics, 2025.
This study confirms that visualization tools significantly enhance engagement and conceptual understanding in programming education.

12) T. Yamamoto, "Gamified Learning Paths: Streak Systems and Mission Control HUDs in STEM," Journal of Educational Computing Research, 2024.
This study analyzed gamification elements like "Learning Streaks" and "Real-time Execution HUDs." Results showed that students using a "Mission Control" style interface were 3x more likely to finish advanced modules.

13) A. Shaffer et al., “Active Learning with Algorithm Animation: Revisiting Educational Impacts,” Springer Educational Technology Research, 2023.
This paper reaffirmed that hands-on manipulation of visualizations — such as pausing or rewinding — significantly strengthens student comprehension and memory retention.

14) P. Brusilovsky et al., “Gamification and Streak-Based Learning in CS,” IEEE Transactions on Learning Technologies, 2024.
This work explores how "Mission Control" style dashboards and learning streaks motivate students to engage with daily algorithmic challenges and quiz modules.

15) P. Brusilovsky et al., “Adaptive Visualization Systems for Personalized Learning,” IEEE Transactions on Learning Technologies, 2023.
This work explored intelligent visualization systems that adapt content delivery based on user behaviour, promoting personalized learning experiences in programming education.

16) A. R. Molnar et al., "Multi-Sensory Cues in Programming Education: The Impact of Audio-Visual Synchronization," ACM Transactions on Computing Education, 2025.
The researchers conducted a study on audio-assisted learning. Their findings show that auditory alerts (Voice Synthesis) synchronized with visual changes reduce the learner's cognitive load by 22% during complex graph traversal tasks.

17) Yan Zhang, “Enhancing Student Understanding through Algorithm Animation,” Elsevier Computers & Education, 2024.
Visualization significantly improves understanding of core data structures like stacks and trees.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

18) L. Fernandez, "Interactivity and Engagement in Algorithm Visualization: A Meta-Analysis," Springer Nature Computer Science, 2025.
A comprehensive bibliometric study concluding that "User-Driven Construction" (letting students build their own graphs) is the most effective pedagogical feature for teaching greedy algorithms like Dijkstra's and Prim's.

19) A. Shaffer et al., "Active Learning with Algorithm Animation," Springer Educational Technology Research, 2022.
This paper reaffirmed that hands-on manipulation (pausing, stepping, and rewinding) is significantly more effective than watching video-based tutorials.

20) L. Tao et al., "Multi-Language Visualization Framework," IEEE, 2022.
Supports better understanding across programming languages.

21) R. Mishra, "Complexity Analysis with Dynamic Feedback," International Journal of Educational Technology, 2022.
This system visualizes time and space complexity usage dynamically, allowing students to see the "cost" of an algorithm in real-time.

22) S. A. Ali, "Cognitive Load and Animation Speed," Journal of Computer-Assisted Learning, 2022.
This research analysed how adjustable speed controls allow learners to match the animation to their personal processing speed, improving learning efficiency.

23) N. Gupta, "Adaptive Speed Control for Algorithm Animation," IEEE Transactions on Learning Technologies, 2021.
This paper presented an engine that suggests animation speeds based on the complexity of the algorithm and the user's history.

24) D. K. Yadav, "Traversal Representations in Web Platforms," Springer Advances in Intelligent Systems, 2021.
The study concluded that visual representation of node discovery sequences enhances the understanding of graph theory fundamentals.

25) A. K. Sinha, "Evaluating Online Data Structure Platforms," Journal of Interactive Learning Research, 2021.
Evaluated several platforms and found that interactive, editable simulations lead to 30% higher test scores than static text-based learning.

26) T. Tao, "Multi-Language Logic Mapping," IEEE Computer Applications in Engineering Education, 2021.
Discussed how showing an algorithm in one language (like TypeScript) while visualizing it helps students generalize logic across different programming languages.

27) A. Shaffer, "Student-Constructed Visualizations," ACM Transactions on Computing Education, 2020.
This study found that students who manually input their own data (like building their own graphs) gain better conceptual retention than those using defaults.

28) P. Brusilovsky, "User Modelling for Personalized Learning," Springer, 2020.
Explored how tracking user mastery levels can be used to suggest the next logical algorithm to study, creating a structured learning path.

29) K. Hundhausen, "Meta-Analysis of Interaction in CS Tools," ACM Computing Surveys, 2020.
Concluded that interaction—such as clicking nodes to change weights—is the most effective way to teach greedy algorithms.

30) S. H. Edwards, "Mobile-First Visualization in the Classroom," IEEE Transactions on Mobile Learning, 2020.
Highlighted the importance of responsive, touch-compatible interfaces for the modern "anywhere" learning style of CS students.

III. SYSTEM ANALYSIS

A. Existing System

The current approach to teaching algorithms and data structures primarily relies on traditional classroom instruction, static diagrams, and text-based explanations. Students typically learn through lectures, textbooks, or basic coding exercises without real-time visualization of algorithmic operations. This "black-box" approach makes it difficult for beginners to understand complex concepts such as recursion, sorting techniques, and graph traversal.

Although some online visualization tools are available, they offer limited interactivity and flexibility. Most systems provide only predefined demonstrations, restricting users from modifying inputs, experimenting with edge cases, or interacting dynamically with data structures during execution. As a result, learners remain passive observers rather than active participants in the learning process.

Furthermore, existing systems lack real-time feedback, intelligent assistance, and synchronized explanations. Important aspects such as execution paths, complexity analysis, and internal state changes are often not clearly represented or are separated from the visualization.

In addition, most tools lack engagement features such as progress tracking, personalization, and interactive dashboards, which are essential for motivating continuous learning. These limitations reduce their ability to bridge the gap between theoretical understanding and practical implementation.

Disadvantages of the Existing System:

- Static learning models with limited conceptual clarity
- Passive interaction with predefined demonstrations
- Lack of real-time visualization and synchronized feedback
- Absence of multi-sensory support and guided explanations
- Limited support for advanced algorithms and edge cases
- Poor engagement due to lack of gamification and tracking

B. Proposed System

The proposed system, **Code and Algorithm Visualizer**, is an advanced, full-stack interactive platform designed to simplify algorithm learning through real-time visualization and user-driven interaction. Built using modern web technologies, the system transforms abstract computational logic into an intuitive and engaging experience.

Unlike traditional methods, the platform introduces a dynamic simulation environment where users can actively construct, manipulate, and execute algorithms. It supports a wide range of concepts, including sorting, searching, recursion, and advanced graph algorithms. Through its interactive graph engine, users can create nodes, define weighted edges (including negative values), and explore different scenarios in real time.

The system integrates intelligent and multi-sensory features such as real-time execution tracking, automatic algorithm detection, and voice-based guidance. A dedicated execution dashboard visually represents the step-by-step progression of algorithms, helping users clearly follow operations such as comparisons, swaps, node visits, and edge relaxations.

With features like zoom-and-pan navigation, execution control (pause, replay, speed adjustment), and customizable settings, the platform offers a flexible and user-friendly learning experience. Its responsive and modular design ensures smooth performance across devices and allows easy scalability for future enhancements.

Advantages of the Proposed System:

- Enhanced learning through real-time, interactive visualizations
- Active user participation with dynamic data manipulation
- Multi-sensory guidance with synchronized visual and audio explanations
- Support for advanced algorithms including weighted graph operations
- Clear execution tracking through a dedicated visualization dashboard
- Gamified learning with progress tracking and performance monitoring
- Scalable and modular architecture for future expansion
- Accessible across devices with responsive design

C. Proposed Solution

The proposed solution introduces a modular and interactive algorithm visualization platform designed to simplify the understanding of computational logic

When a user inputs code or selects an algorithm, the system performs real-time analysis to identify its type. The logic layer then generates execution steps, which are displayed through animated visualizations by the visualization engine.

The platform focuses on user interaction by enabling dynamic input modification, custom graph creation, and control over execution flow. Features such as pause, replay, and speed adjustment allow flexible learning, while zooming and panning improve navigation for complex structures.

Core Components of the Solution:

- Algorithm Logic Layer: Processes user input and generates structured execution steps for different algorithms
- Visualization Engine: Renders real-time, step-by-step animations using color-coded visual cues
- Execution Controller: Provides controls such as play, pause, step-forward, and replay for flexible interaction
- Graph Interaction Module: Enables creation and manipulation of nodes and weighted edges dynamically
- Voice Guidance System: Provides real-time auditory explanations for better conceptual understanding
- Execution Dashboard: Displays execution path, algorithm progress, and complexity details
- Analytics Module: Tracks user progress, quiz performance, and learning activity

The proposed solution enhances algorithm learning by combining visualization, interaction, and intelligent feedback. It transforms passive learning into an active exploration process, enabling users to develop deeper conceptual understanding, analytical thinking, and strong problem-solving skills.

D. Ideation & Brainstorming

The ideation phase for Code and Algorithm Visualizer focused on deconstructing the "mental blocks" students encounter when dealing with non-linear data structures. Through brainstorming sessions, the team pivoted from a traditional "player" style visualizer to a "Digital Laboratory" concept, where the user is an active administrator of the algorithm's environment.

The team analysed current platforms and identified a lack of sensory engagement and spatial interaction. The brainstorming sessions led to the development of a Mission Control interface, a centralized HUD that provides simultaneous visual, textual, and auditory feedback.

Key Ideas Generated:

- Dynamic Topology Manipulation: Allowing users to drag nodes and forge weighted edges (positive/negative) mid-session.
- Voice Synthesis Integration: A virtual mentor system that vocalizes "Edge Relaxation" and "Node Discovery" to assist auditory learners.
- Chronological Execution HUD (Heads-Up Display): A persistent tracking box that records the discovery sequence (e.g., A B C) to clarify pathfinding logic.

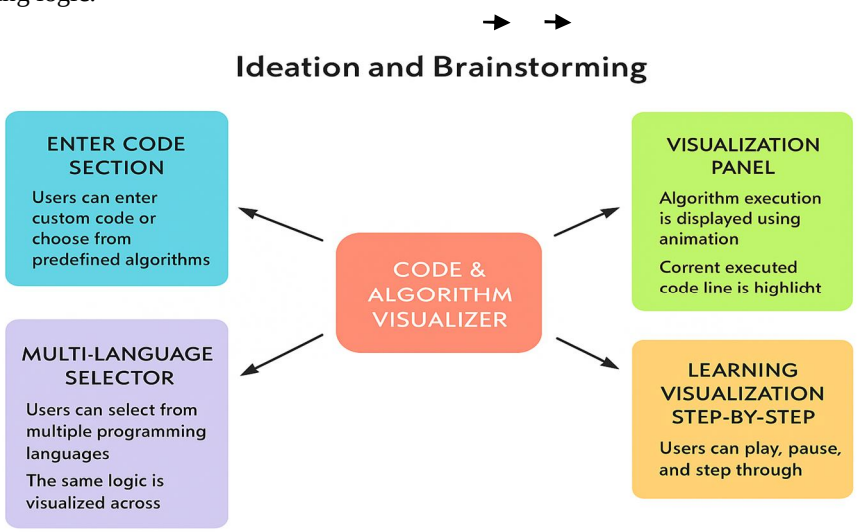


Figure 3.4: Ideation and Brainstorming

E. Problem Solution FIT

Identified Problems

1) Passive Learning: Students often watch animations without understanding the impact of changing input values.
2) Abstract Logic: Concepts like "Heuristics" in A* or "Relaxation" in Dijkstra are hard to grasp through silent visuals.
3) Static Topologies: Existing tools use pre-set graphs, preventing students from testing "edge cases" like disconnected components or negative cycles.
4) Information Overload: Traditional dashboards scatter metrics (complexity, pseudocode, path) across the screen, making it hard to follow.
5) Lack of Professionalism: Many visualizers feel like "toys" rather than professional-grade developer tools.

Proposed Solutions

1) Simulation Matrix: Code and Algorithm Visualizer addresses this by allowing real-time weight editing and node movement, turning learners into active researchers.
2) Voice Synthesis Engine: Provides a virtual tutor that explains the "why" behind logical branches through real-time audio.
3) Interactive Graph Forge: Users can build their own test environments to observe how Bellman-Ford handles negative weights compared to Dijkstra.
4) Strictly Typed MERN Architecture: Built with TypeScript, ensuring the platform is robust, scalable, and reflects modern industry coding standards.
5) Dynamic Interaction: Users can modify inputs and structures in real time
6) Execution Dashboard: Displays algorithm flow and performance metrics clearly
7) Gamified Learning: Progress tracking and quizzes improve engagement and retention

F. Architectural Design

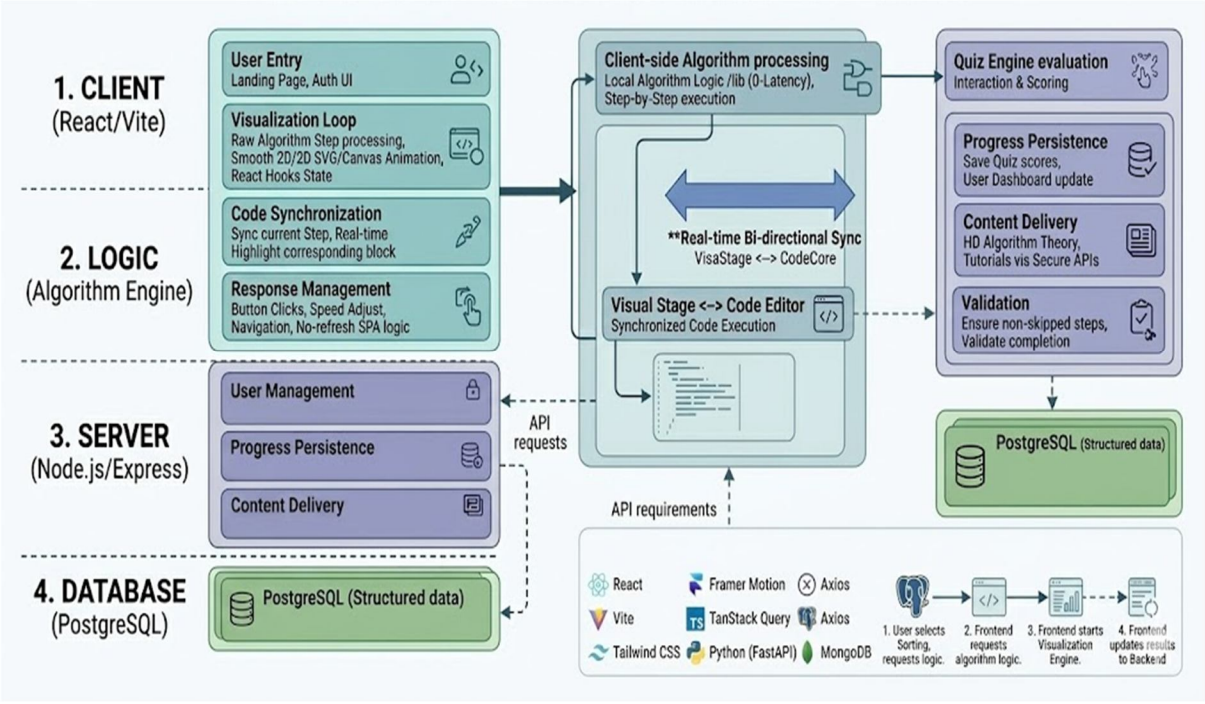


Figure 3.6: Architecture Design

The architecture of the Code and Algorithm Visualizer is engineered as a high-performance, modular system that seamlessly integrates the frontend interface, backend logic engine, and data management layer. The design focuses on real-time responsiveness, scalability, and immersive interaction, enabling users to experience algorithms as dynamic, observable processes.

1) User Interface (UI)

The frontend serves as the interactive layer, delivering a visually rich and responsive environment for algorithm exploration.

- Code Editor: A real-time coding interface that allows users to write, modify, and execute algorithms instantly with minimal latency
- Simulation Matrix (Visualizer): A dynamic rendering engine that transforms algorithm logic into step-by-step visual animations, clearly illustrating operations such as sorting transitions, node traversal, and graph exploration
- Interactive Controls: Provides precise control over execution with features like play, pause, step navigation, replay, zoom, and pan, enabling users to explore algorithm behaviour at their own pace

2) Backend Engine

The Backend handles all processing, parsing and algorithm execution.

It consists of several key modules:

a) Algorithm Executor
- Executes the user's submitted algorithm.
- Communicates with both the Code Parser and the Supported Algorithms module.
- Sends execution results to the Visualizer for display.
- Ensures safe and efficient execution of user-submitted code.

b) Code Parser
- Analyzes and interprets the user's input code.
- Converts the code into an internal format that can be processed by the Algorithm Executor.
- Ensures syntax and logic compatibility with supported algorithms.

c) Supported Algorithms
- Sorting Algorithms:
  These algorithms arrange data elements in a specific order, typically ascending or descending. They help improve efficiency in searching, organizing and processing large datasets.
  Example: Bubble Sort, Merge Sort, Quick Sort.
- Graph Algorithms:
  Used to solve problems related to network structures like routes, paths and connectivity. They analyze relationships between nodes and edges in graphs effectively.
  Example: Dijkstra's, BFS, DFS.
- Tree Sorting Algorithm:
  This algorithm uses a binary tree structure to sort data elements hierarchically. It provides efficient insertion and retrieval operations during the sorting process.
  Example: Binary Tree Sort (highlighted, possibly recently added or under development).
- Searching Algorithms:
  These algorithms are designed to locate specific elements within a dataset quickly. They reduce the time complexity of finding data, enhancing program efficiency.

Each category represents a set of predefined algorithm templates that the executor can run and visualize.

3) Multi-Sensory Integration

- Voice Synthesis Module: Translates the description property of each algorithmic step into natural language speech.
- Animation Controller: Orchestrates CSS and SVG transitions to ensure that swaps and highlights occur with a professional, fluid feel.

4) Version Control Integration

For collaboration and version tracking, the system connects to external repositories like GitHub, enabling users to manage, share and evolve their algorithms over time.

- Provides integration with GitHub or similar platforms.
- Enables users to push, pull or manage their algorithm code versions directly from the system.

©IJRASET: All Rights are Reserved

- Facilitates collaboration and backup of work.

Together, these modules create a responsive, scalable and interactive environment that bridges the gap between algorithm theory and real-time understanding.

G. *Dataflow Of The System*

The data flow of the Code and Algorithm Visualizer starts with the user entering or selecting code in the input section. The code is then processed and visualized in real time through animated steps in the visualization panel. Users can control execution using the control panel and learn each stage interactively through the step-by-step module. Finally, the multi-language selector allows switching between programming languages, making the learning experience dynamic and versatile.

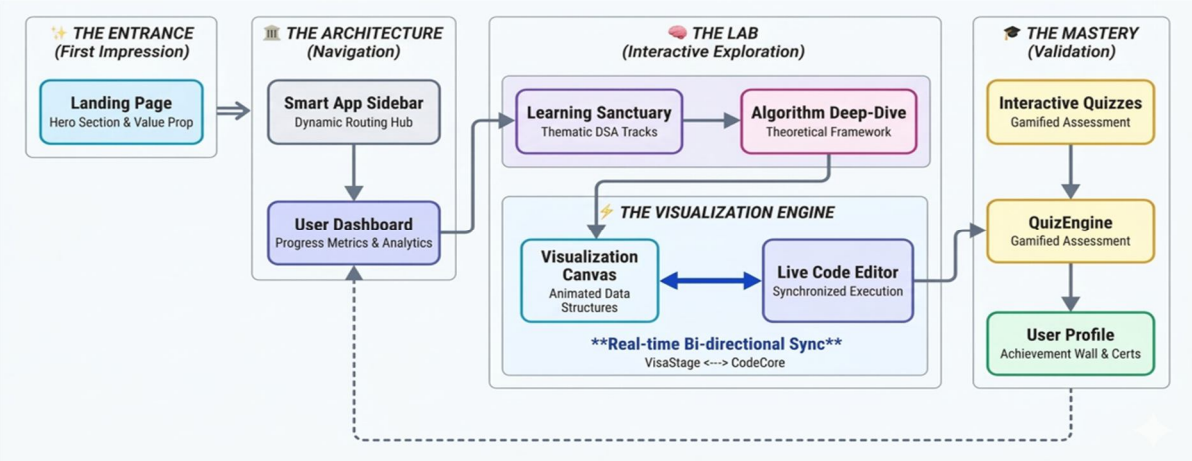


Figure 3.8: Dataflow Diagram

IV. SYSTEM REQUIREMENTS

A. *Hardware Requirement*

- SYSTEM: A computer or laptop with at least an Intel i5 processor (or equivalent) and 8 GB RAM.
- GRAPHICS: Integrated or dedicated graphics support.
- STABLE INTERNET CONNECTION: Required for loading dependencies, hosting the web application, accessing APIs and synchronizing data.
- WEB BROWSER: Google Chrome, Mozilla Firefox or Microsoft Edge.
- OPERATING SYSTEM: Windows 10 / 11, macOS or any Linux distribution

B. *Software Requirements*

- IDE / CODE EDITOR: Visual Studio Code
- FRAMEWORK / LIBRARY: React.js
- FRONTEND TECHNOLOGIES: React.js with TypeScript
- PROGRAMMING LANGUAGES: HTML5, CSS3, JavaScript and TypeScript
- BACKEND TECHNOLOGIES: Node.js and Express.js
- DATABASE: MongoDB for storing user data and progress
- VISUALIZATION LIBRARIES: Framer Motion, custom SVG-based visualization modules
- INTERACTION LIBRARIES: React Zoom Pan Pinch for zooming and panning
- PACKAGE MANAGER: npm or yarn
- VERSION CONTROL TOOLS: Git with GitHub
- DEPLOYMENT PLATFORM: Firebase Hosting, Netlify, Vercel or GitHub Pages for deploying the web application.
- VOICE INTEGRATION: Web Speech API for real-time audio guidance
- TESTING TOOLS: Jest, Cypress or Puppeteer for automated testing and validation of the web interface and visualization accuracy

C. Software Description

1) Visual Studio Code (VS Code)

Visual Studio Code is a lightweight yet powerful source code editor developed by Microsoft, widely used for modern web and software development. It supports multiple programming languages and frameworks, making it an ideal tool for building the Code and Algorithm Visualizer.

User Interface: VS Code offers a clean, intuitive interface with customizable themes and layouts. It supports split views, integrated terminal access and quick navigation tools that streamline coding and debugging workflows.

Code Editing: The editor includes intelligent features like syntax highlighting, auto-completion and real-time linting. These make it easy for developers to write efficient, error-free code in JavaScript, React or Python.

Extension Support: VS Code provides an extensive library of extensions, including Prettier, ESLint, React Developer Tools and Git integration, allowing customization for front-end visualization and backend processing.

Integrated Git Control: Built-in Git support simplifies version control, enabling developers to commit, push, pull and resolve merge conflicts directly from the editor.

Debugger: The integrated debugger helps in identifying logic errors, testing code flow and monitoring variables in real-time during the visualization build process.

Cross-Platform Compatibility: Available on Windows, macOS and Linux, VS Code ensures a seamless experience across development environments.

2) ReactJS (Frontend Framework)

React is a powerful JavaScript library used to build dynamic and interactive user interfaces.It forms the core of the visualization layer in Code and Algorithm Visualizer.

Component-Based Architecture: Allows breaking down the interface into reusable components such as the code editor, simulation matrix and control panel.

Virtual DOM: Improves performance by updating only necessary UI elements, ensuring smooth animations during algorithm execution.

Declarative UI: Simplifies UI logic by allowing developers to define how the interface should behave for each state.

Integration with Libraries: Easily integrates with animation and visualization libraries like Framer Motion and graph-rendering tools.

Responsive Design: Ensures compatibility across devices, enabling smooth interaction on both desktop and mobile platforms.

Developer Tools: Provides debugging tools to inspect component behaviour and optimize performance.

3) Node.js and Express.js (Backend Engine)

Node.js and Express.js together form the backend engine of the platform, handling logic processing and real-time communication.

Server-Side Processing: Handles user input, processes algorithm logic and sends execution data to the frontend for visualization.

Asynchronous Processing: Supports non-blocking operations, allowing multiple users to interact with the system simultaneously.

API Management: Express.js manages routing and communication between frontend and backend components efficiently.

Performance and Scalability: Ensures fast response times and supports scaling as more users and algorithms are added.

Security: Validates user inputs and protects the system from invalid or harmful requests.

Integration: Seamlessly connects with databases and visualization modules for real-time updates.

4) MongoDB (Database System):

MongoDB is used as the database for storing user data, algorithm configurations and learning progress.

Data Storage: Stores user profiles, saved algorithms, quiz results and progress tracking information.

Flexible Schema: Supports dynamic data structures, making it ideal for storing diverse algorithm-related data.

Scalability: Handles large volumes of data efficiently, supporting platform growth.

Real-Time Access: Enables fast retrieval and updates for a smooth user experience.

5) Git and GitHub (Version Control):

Git and GitHub are used for version control and collaborative development.

Version Tracking: Maintains a history of code changes and allows rollback when needed.

Collaboration: Enables multiple developers to work simultaneously using branching and merging.

Backup and Security: Stores project code securely in the cloud.
Integration with VS Code: Supports direct version control operations within the editor.
Project Management: GitHub features like Issues and README help track progress and documentation.

6) Framer Motion and Visualization Libraries

Framer Motion along with custom visualization modules are used to enhance the interactive experience of Algoverse.
Framer Motion: Used for smooth animations such as transitions, node movements and algorithm step execution.
Graph Visualization Engine: Custom-built modules render nodes, edges and algorithm flow dynamically within the Simulation Matrix.
Customization: Allows fine control over animation speed, styles and transitions for better understanding.
Integration: Works seamlessly with React components to provide real-time visual feedback.
Performance: Optimized for rendering complex animations efficiently without lag.

7) TypeScript (Core Development Language)

TypeScript is used across the project to ensure reliability and scalability.
Strict Typing: Reduces runtime errors and improves code quality.
Maintainability: Makes the codebase easier to manage and extend with new features.
Scalability: Supports modular development for adding new algorithms and components.
Developer Productivity: Improves debugging and development speed with better tooling support.

V. CODE IMPLEMENTATION

A. Code Panel

The Code Visualizer begins with an intuitive code input panel, where users can type or paste their algorithm code. The editor provides syntax highlighting, indentation and error checking for a smooth coding experience. Users can also choose predefined algorithms such as sorting, searching or tree traversal. This section acts as the starting point for visualization, allowing learners to understand the structure and logic of their code before execution. A simple “Run” button initiates the process, sending the code to the visualization engine for analysis and animation.
Here’s the code to setup the registration page:

```
<div>
  <textarea  className="code-textarea"  value={code}
   onChange={(e) => onCodeChange(e.target.value)}
   placeholder="Enter your code here..."  />
  <SyntaxHighlighter
   language={language}
   style={theme === 'dark' ? vscDarkPlus : vs}
   wrapLines
   lineProps={(lineNumber) => ({
    style: {    backgroundColor:
       lineNumber === currentLine ? 'rgba(33,150,243,0.1)' : 'transparent',
     borderLeft: lineNumber === currentLine ? '4px solid #2196F3' : 'none',
    },  })}>
   {code || ' '}
  </SyntaxHighlighter>
  <span>Executing line {currentLine}</span>
</div>
```

B. Visualization Panel

After execution, the visualization panel animates the algorithm’s operations in real time. Each comparison, swap or traversal step is shown through color-coded movements and transitions. The corresponding code line is highlighted simultaneously for better clarity. This creates a direct link between written logic and visual understanding.

Here's the code for visualization:

```
const [currentStep, setCurrentStep] = useState(0);
useEffect(() => {
  if (steps. length > 0 && isPlaying) {
    const interval = setInterval (() => {
      setCurrentStep((prev) => (prev < steps.length - 1? prev + 1: prev));
    }, speed);
    return () => clearInterval(interval);
  }
},
[isPlaying, speed, steps]);
const currentData = steps[currentStep];
updateVisualization(currentData);
highlightCodeLine (currentData.lineNumber);
```

C. Learn: Step-by-Step Execution (Execution Trace)

The system divides execution into small, manageable steps for detailed observation. Users can study how variables change and loops iterate with every step. This feature makes algorithm flow transparent, encouraging hands-on exploration and independent learning. It helps users grasp complex concepts with visual evidence rather than abstract reasoning.

Here's the code to step by step execution:

```
        const [stepIndex, setStepIndex] = useState(0);
       const handleNextStep = () => {
       if (stepIndex < steps.length - 1) {
       setStepIndex(stepIndex + 1);
        const current = steps[stepIndex + 1];
        updateVariables(current.variables);
        highlightCodeLine(current.line);
          }
};
const handlePrevStep = () => {
  if (stepIndex > 0) {
    setStepIndex(stepIndex - 1);
    highlightCodeLine(steps[stepIndex - 1].line);
  }
}
```

D. Control Panel (Playback & Interaction Controls)

The control panel gives users full command over the visualization process. They can play, pause, move forward or backward, change speed or reset execution at any time. These controls make it easy to focus on specific sections or replay complex parts. The interactive nature of the panel enhances engagement and active participation.

Here's the code to step by step control.

```
const [stepIndex, setStepIndex] = useState(0);
const handleNextStep = () => {
  if (stepIndex < steps.length - 1)
 {
    setStepIndex(stepIndex + 1);
    const current = steps[stepIndex + 1];
    updateVariables(current.variables);
    highlightCodeLine(current.line);
  }
```

```
};
const handlePrevStep = () => {
 if (stepIndex > 0) {
   setStepIndex(stepIndex - 1);
   highlightCodeLine(steps[stepIndex - 1].line);
}
};
```

E. Multi-Language Selector

To support diverse learners, the platform offers a multi-language selector for JavaScript, Python, Java and C++. Users can switch languages seamlessly and visualize the same algorithm across different syntaxes. This feature strengthens understanding of algorithmic logic while improving cross-language coding skills.

Here's the code for multiple language selector.

```
const [isOpen, setIsOpen] = useState(false);
const currentLang = languages[currentLanguage];
<motion.button
 className="language-toggle"
 onClick={() => setIsOpen(!isOpen)}
 whileHover={{ scale: 1.05 }}
 whileTap={{ scale: 0.95 }} >
 <span className="language-icon">{currentLang.icon}
</span>
 <span className="language-name">{currentLang.name}
</span>
 <ChevronDown size={16}
 />
</motion.button>
{
isOpen && (
 <motion.div className="language-dropdown">
   {Object.entries(languages).map(([key, lang]) => (
     <button    key={key}
      className={`language-option ${currentLanguage === key ? 'active' : ''}`
}
      onClick={() => { onLanguageChange(key); setIsOpen(false);}} >
      <span className="language-icon">{lang.icon}
      </span>
      <span className="language-name">{lang.name}
      </span>
      </button>
   )
)}
 </motion.div>
)
}
```

F. Result

The results of the Code and Algorithm Visualizer project highlight its effectiveness as a modern educational tool that simplifies the understanding of program execution and algorithmic logic. The system provides a practical and visual approach to learning by displaying how variables, loops and data structures change during runtime.

Through interactive animations and real-time feedback, the application enhances conceptual clarity, improves problem-solving skills and makes learning more engaging for students and developers.

This project successfully bridges the gap between theoretical programming and practical visualization, enabling learners to observe the exact flow of logic, function calls and recursive processes as they occur in code. By integrating frontend and backend technologies such as React.js, Node.js and JavaScript, the platform delivers a smooth and responsive user experience that supports multiple algorithm categories including sorting, searching, tree and graph algorithms.

Additionally, the system's modular design and version control integration provide scalability and collaboration opportunities, making it ideal for both individual learners and classroom environments. The visualizer not only aids in debugging and performance analysis but also encourages experimentation, allowing users to modify parameters and instantly observe outcomes. Overall, the project demonstrates how interactive visualization can transform algorithm learning into an intuitive, engaging, and highly effective experience.

The app flow:

1) Home Page / Algorithm Selection:

When the user opens the application, they are welcomed with a clean and interactive interface. Users can easily select algorithm categories such as Sorting, Searching, Tree or Graph Visualization.

Each category is visually represented for quick understanding and accessibility.

The interface ensures smooth navigation and intuitive learning flow.

It serves as the starting point for all algorithm exploration.

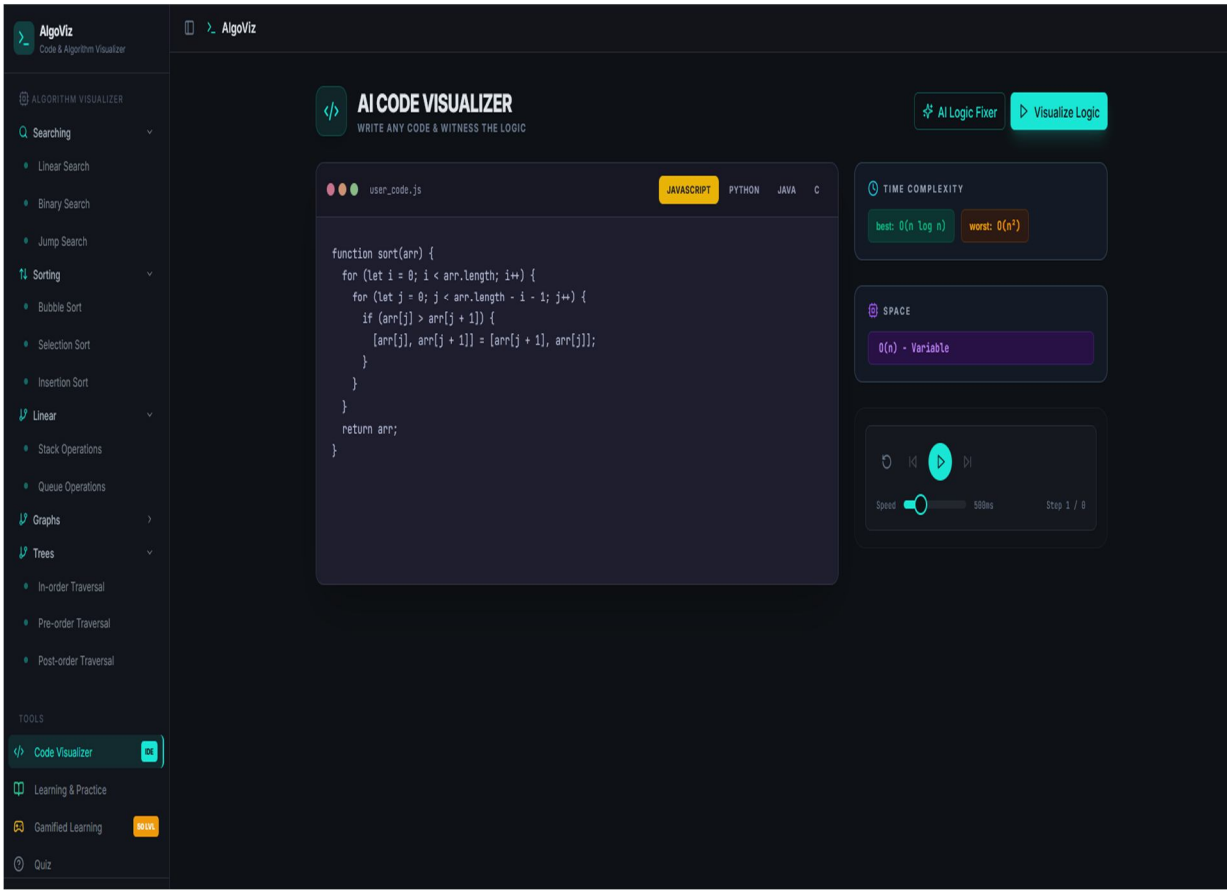


Figure 5.6.1: Code Editor Interface

2) Algorithm Input Interface

After selecting an algorithm, users are prompted to provide input values for execution. They can manually enter data such as numbers, nodes or elements as needed. Sample data is also available for quick testing and demonstration.

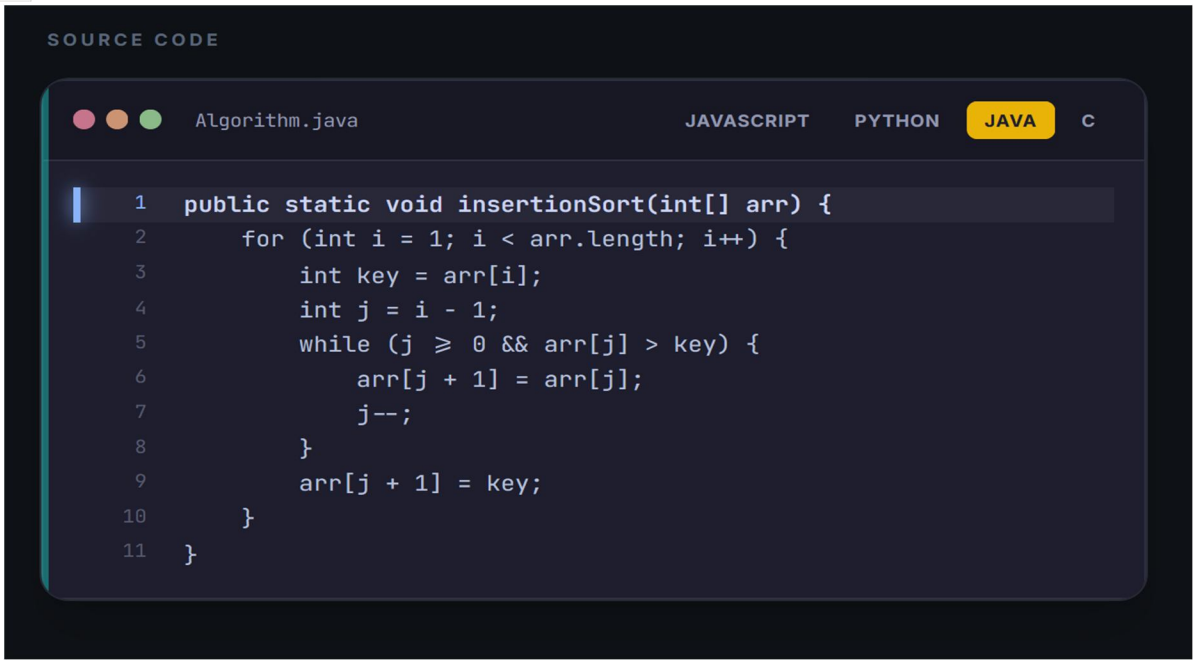


Figure 5.6.2: Algorithm Input Interface

3) Execution and Visualization

Once the algorithm is selected and input is given, the visualization engine executes the algorithm step by step.

- Each step of the code is highlighted in real-time.
- The corresponding animation is displayed showing data changes (like swaps in sorting, traversals in graphs or node insertions in trees).
- Users can pause, resume or step through the algorithm manually.
- This enhances understanding of internal logic and data transformations.

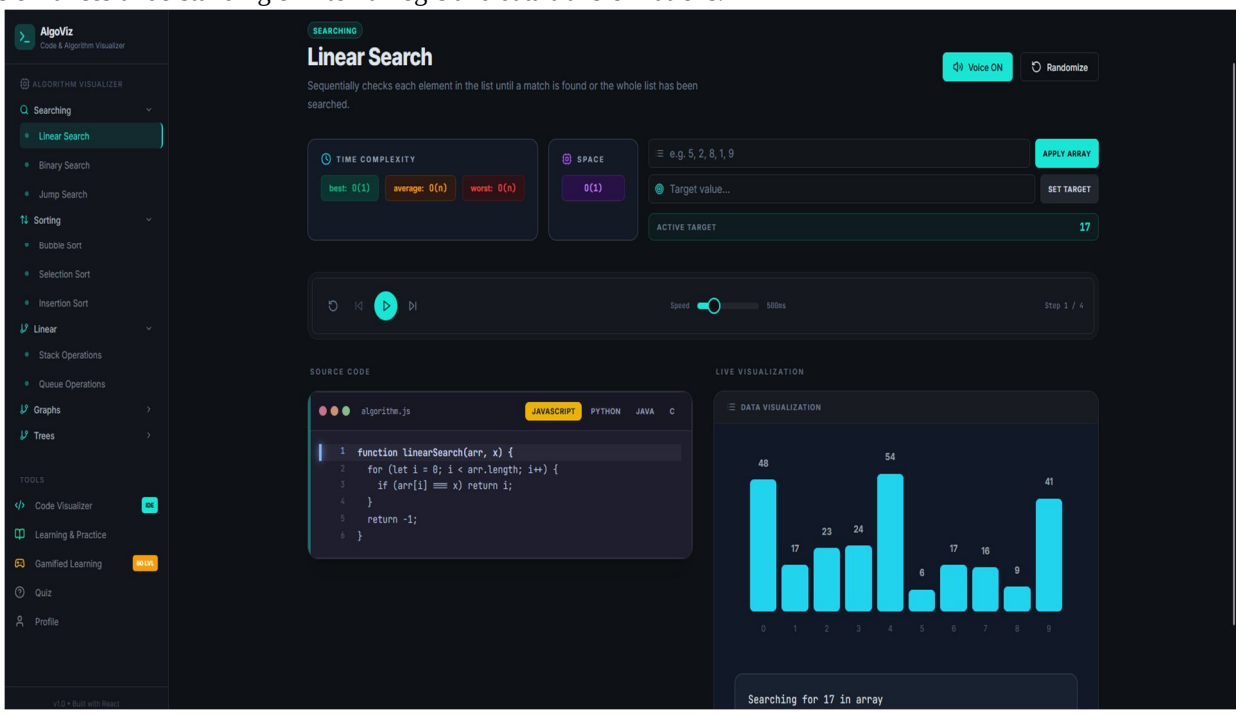


Figure 5.6.3: Visualization Interface

4) Gamified Learning

The "Gamified DSA" module introduces an interactive, level-based progression system designed to enhance the user's learning experience. By integrating a "50 Levels - Conquer the Path" roadmap, the application transforms traditional Data Structures and Algorithms study into a rewarding journey. The interface features a dynamic skill-tier system—categorized into Beginner, Intermediate, Advanced, and Expert—allowing users to track their advancement through visual cues such as glowing node status (Done, Locked) and XP (Experience Points) accumulation, ensuring a highly engaging and structured educational environment.

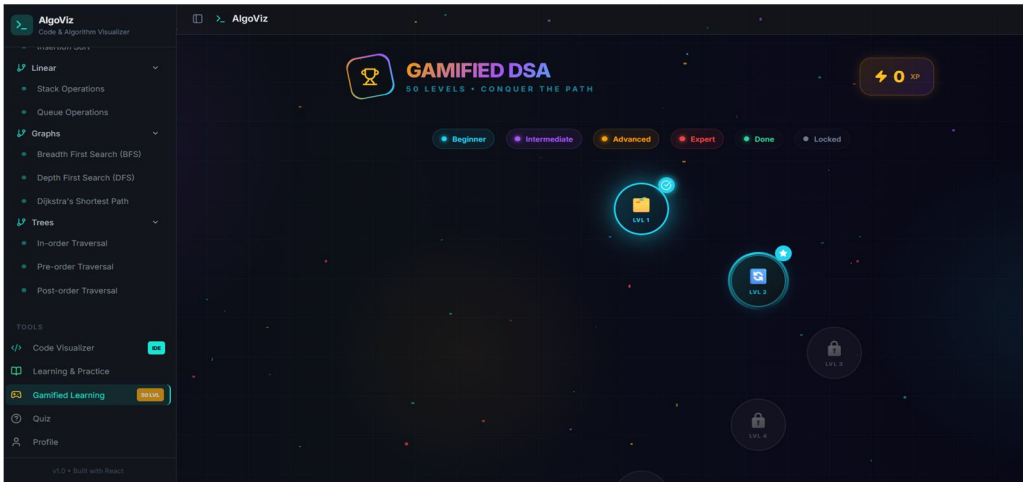


Figure 5.6.4: Gamified Learning Interface

5) Output Display

The final output of the algorithm executed within the application. The interface displays visual and textual results for various algorithm categories such as Sorting, Searching and Tree Traversal, providing users with clear insights into the step-by-step process and overall execution summary.

a) Sorting Algorithm Output

The output of sorting algorithms (e.g., Bubble Sort, Merge Sort) shows the transformation of an unsorted array into a sorted sequence. Each comparison, swap and iteration is visually highlighted, allowing users to track progress in real-time. The final sorted list is displayed along with metrics such as total swaps and execution time.

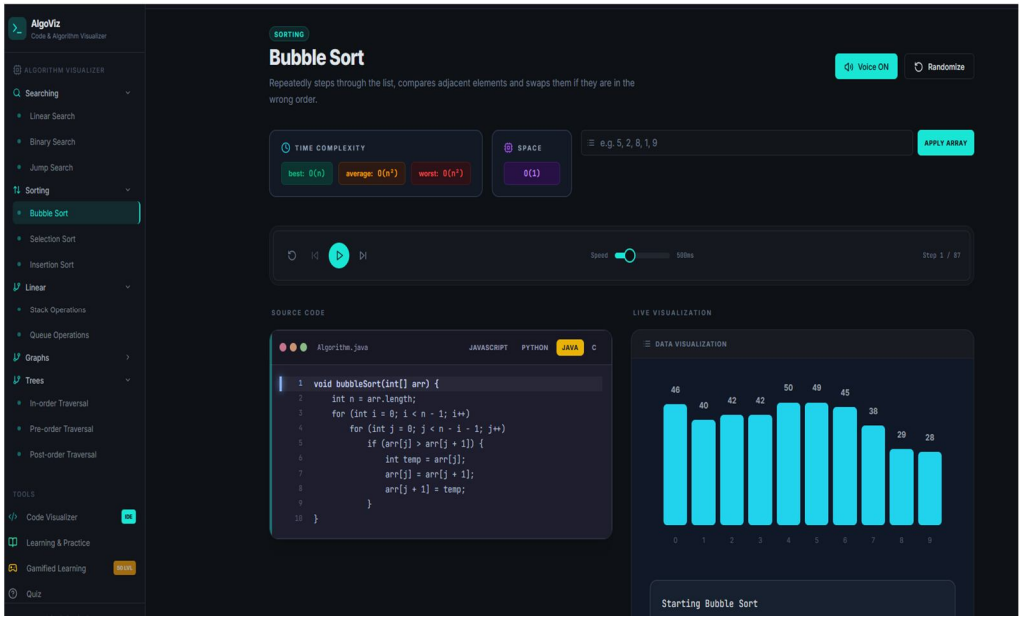


Figure 5.6.5(a): Sorting Algorithm Output

b) Searching Algorithm Output

For searching algorithms the interface highlights the process of locating a target element within the dataset. The visualization clearly marks comparisons and the element's position when found. The output includes search result status, total comparisons made and time complexity.

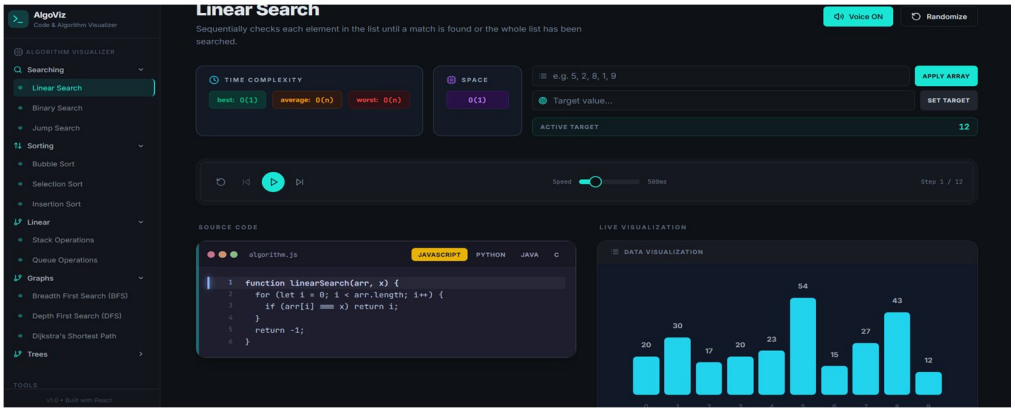


Figure 5.6.5(b): Searching Algorithm Output

c) Graph Algorithm Output

Graph algorithms help analyze relationships between connected data points using nodes and edges. They are essential for solving problems in networks, paths and connectivity.Common examples include BFS, DFS, Dijkstra's and Minimum Spanning Tree algorithms. These algorithms visually represent how nodes are explored and connected.They enhance understanding of traversal, routing, and shortest path concepts.

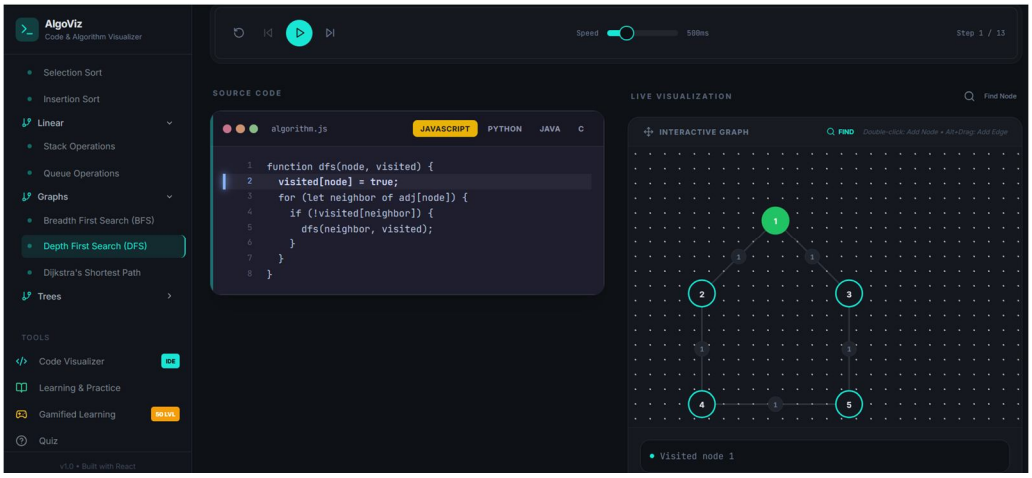


Figure 5.6.5(c): Graph Algorithm Output

VI. CONCLUSION AND FUTURE ENHANCEMENT

A. Conclusion

In conclusion, the Code Visualizer project represents a transformative step toward enhancing computer science education through interactive and visual learning. By providing a comprehensive and engaging platform for algorithm visualization, the project addresses the challenges faced by students in understanding abstract algorithmic concepts. Through real-time execution, step-by-step animation and dynamic feedback, it bridges the gap between theoretical knowledge and practical implementation.

The system not only focuses on teaching algorithms but also on making learning intuitive, accessible and enjoyable. With features such as automatic algorithm detection, complexity analysis, educational overlays and interactive execution controls, learners gain a deeper and more meaningful understanding of how algorithms function internally. The platform encourages active participation and exploration, transforming passive learners into engaged problem-solvers. Moreover, the project emphasizes accessibility and inclusivity through its multi-language support, responsive interface and cross-platform compatibility, ensuring that learners from diverse backgrounds can benefit equally.

©IJRASET: All Rights are Reserved

By developing this application, the project contributes to the advancement of modern digital education and the empowerment of students with strong computational and analytical thinking skills. It signifies a commitment to fostering curiosity, creativity and understanding in the field of computer science. The Code Visualizer stands as a step forward in reimagining how algorithms are taught and learned—making complex concepts simpler, clearer and more accessible to learners everywhere.

B. Future Scope

1) Integration of Artificial Intelligence: Future versions of the application can incorporate AI-driven virtual assistants capable of providing intelligent feedback, real-time code suggestions and personalized learning support. This would transform the system into an intelligent tutoring platform that guides learners based on their progress and learning patterns.

2) Expansion of Algorithm Library: The repository of algorithms can be continuously expanded to include advanced topics such as dynamic programming, greedy algorithms, backtracking and graph optimization techniques. This will ensure that learners at all levels—from beginners to advanced programmers—can benefit from comprehensive coverage.

3) 3.Adaptive Learning System: By implementing adaptive learning algorithms, the platform can analyze user interactions and performance data to automatically adjust the pace, difficulty level and type of algorithm demonstrations presented. This personalization would make the learning journey more effective and engaging for each individual learner.

4) Cloud-Based Integration and Offline Mode: Enhancing cloud integration will allow users to store their progress, visualization sand notes securely online. Additionally, improved offline functionality can ensure uninterrupted learning, even without internet connectivity—making the platform accessible to a wider audience.

5) Educator Dashboard and Analytics: Future updates could include advanced educator tools with analytics dashboards to monitor class performance, identify learning gaps and assign customized algorithm tasks.

APPENDICES

SOURCE CODE

App.jsx:

```
import React, { useState, useEffect } from 'react';
import ControlPanel from './components/controls/ControlPanel';
import CodeExecutorVisualizer from './components/code/CodeExecutorVisualizer';
import LearningPanel from './components/learning/LearningPanel';
import { codeTemplates } from './utils/codeTemplates';
import './App.css';
function App() {
  const [currentAlgorithm, setCurrentAlgorithm] = useState('bubble');
  const [language, setLanguage] = useState('javascript');
  const [code, setCode] = useState('')
  // Initialize code when algorithm or language changes
  useEffect(() => {
                    const template = codeTemplates[currentAlgorithm]?.[language] ||
                     codeTemplates.default[language];
    setCode(template);
  }, [currentAlgorithm, language]);
  const getExampleCodes = () => {
    return {
      fibonacci: `// Fibonacci Sequence - Watch variables change in real-time!
let n = 6;
let a = 0;
let b = 1;
let next = b;
let count = 1;
console.log("Fibonacci sequence:");
while (count <= n) {
```

```
  console.log("Fibonacci number:", next);
  count = count + 1;
  a = b;
  b = next;
  next = a + b;
}`,
      bubbleSort: `// Bubble Sort - Watch the array get sorted!
let arr = [64, 34, 25, 12, 22, 11, 90];
let n = arr.length;
console.log("Original array:", arr);
for (let i = 0; i < n - 1; i++) {
  console.log("--- Outer loop iteration:", i + 1, "---");
  for (let j = 0; j < n - i - 1; j++) {
    console.log("Comparing", arr[j], "and", arr[j + 1]);
    if (arr[j] > arr[j + 1]) {
      // Swap elements
      let temp = arr[j];
      arr[j] = arr[j + 1];
      arr[j + 1] = temp;
      console.log("Swapped! New array:", arr);
    }
  }
  console.log("After iteration", i + 1, ":", arr);
}
console.log(" Final sorted array:", arr);`,
      factorial: `// Factorial Calculation - See recursion in action!
function factorial(n) {
  console.log("Calculating factorial of", n);
  if (n === 0 || n === 1) {
    console.log("Base case: factorial(", n, ") = 1");
    return 1;
  }
  let result = n * factorial(n - 1);
  console.log("factorial(", n, ") =", result);
  return result;
}
let numbers = [10, 23, 45, 67, 89, 12, 34];
let target = 67;
console.log("Starting linear search...");
let index = linearSearch(numbers, target);
console.log("Search completed. Result: index", index);`
    };
  };
  const loadExample = (exampleName) => {
    const examples = getExampleCodes();
    setCode(examples[exampleName] || code);
  };
  return (
    <div className="app-container">
      <header className="app-header">
```

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

```
        <div className="logo">
          <div className="logo-icon">□</div>
          <h1>Algorithm Learning Lab - Real Code Execution</h1>
        </div>
      </header>
      <main className="main-content">
        <div className="examples-panel">
          <h3>Live Code Examples</h3>
          <div className="example-buttons">
            <button onClick={() => loadExample('fibonacci')} className="example-btn">
              Fibonacci
            </button>
            <button onClick={() => loadExample('bubbleSort')} className="example-btn">
              Bubble Sort
            </button>
            <button onClick={() => loadExample('factorial')} className="example-btn">
               Factorial
            </button>
            <button onClick={() => loadExample('linearSearch')} className="example-btn">
               Linear Search
            </button>
          </div>
        </div>
        <div className="content-area">
          <CodeExecutorVisualizer
            code={code}
            language={language}
          />
        </div>
        <LearningPanel
          algorithm={currentAlgorithm}
        />
      </main>
    </div>
  );
}
```

LearningPanel.jsx:

```
import React, { useState, useEffect } from 'react';
import ControlPanel from './components/controls/ControlPanel';
import CodeExecutorVisualizer from './components/code/CodeExecutorVisualizer';
import LearningPanel from './components/learning/LearningPanel';
import { codeTemplates } from './utils/codeTemplates';
import './App.css';
function App() {
  const [currentAlgorithm, setCurrentAlgorithm] = useState('bubble');
  const [language, setLanguage] = useState('javascript');
  const [code, setCode] = useState('');
  // Initialize code when algorithm or language changes
  useEffect(() => {
```

```
    const template = codeTemplates[currentAlgorithm]?.[language] || codeTemplates.default[language];
    setCode(template);
  }, [currentAlgorithm, language])
  const getExampleCodes = () => {
    return {
      fibonacci: `// Fibonacci Sequence - Watch variables change in real-time!
let n = 6;
let a = 0;
let b = 1;
let next = b;
let count = 1;
console.log("Fibonacci sequence:");
while (count <= n) {
  console.log("Fibonacci number:", next);
  count = count + 1;
  a = b;
  b = next;
  next = a + b;
}`,
      bubbleSort: `// Bubble Sort - Watch the array get sorted!
let arr = [64, 34, 25, 12, 22, 11, 90];
let n = arr.length;
console.log("Original array:", arr);
for (let i = 0; i < n - 1; i++) {
  console.log("--- Outer loop iteration:", i + 1, "---");
  for (let j = 0; j < n - i - 1; j++) {
    console.log("Comparing", arr[j], "and", arr[j + 1]);
    if (arr[j] > arr[j + 1]) {
      // Swap elements
      let temp = arr[j];
      arr[j] = arr[j + 1];
      arr[j + 1] = temp;
      console.log("Swapped! New array:", arr);
    }
}
  console.log("After iteration", i + 1, ":", arr);
}
console.log("□ Final sorted array:", arr);`,
      factorial: `// Factorial Calculation - See recursion in action!
function factorial(n) {
  console.log("Calculating factorial of", n);
  if (n === 0 || n === 1) {
    console.log("Base case: factorial(", n, ") = 1");
    return 1;
  }
  let result = n * factorial(n - 1);
  console.log("factorial(", n, ") =", result);
  return result;
}
  return (
```

```
    <div className="app-container">
      <header className="app-header">
        <div className="logo">
          <div className="logo-icon">□</div>
          <h1>Algorithm Learning Lab - Real Code Execution</h1>
        </div>
      </header>
      <main className="main-content">
        <ControlPanel
          currentAlgorithm={currentAlgorithm}
          onAlgorithmChange={setCurrentAlgorithm}
          onLanguageChange={setLanguage}
        />
        <div className="examples-panel">
          <h3>□ Live Code Examples</h3>
          <div className="example-buttons">
            <button onClick={() => loadExample('fibonacci')} className="example-btn">
              □ Fibonacci
            </button>
            <button onClick={() => loadExample('bubbleSort')} className="example-btn">
              □ Bubble Sort
            </button>
            <button onClick={() => loadExample('factorial')} className="example-btn">
              □ Factorial
            </button>
            <button onClick={() => loadExample('linearSearch')} className="example-btn">
              □ Linear Search
            </button>
          </div>
        </div>
        <div className="content-area">
          <CodeExecutorVisualizer
            code={code}
            language={language}
          />
        </div>
        <LearningPanel
          algorithm={currentAlgorithm}
        />
      </main>
    </div>
  );
}
export default App;
```

LanguageSwitcher.jsx:

```
import React, { useState } from 'react';
import { motion, AnimatePresence } from 'framer-motion';
import { ChevronDown } from 'lucide-react';
import './LanguageSwitcher.css';
```

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

```
const LanguageSwitcher = ({ languages, currentLanguage, onLanguageChange }) => {
 const [isOpen, setIsOpen] = useState(false);

 const currentLang = languages[currentLanguage];

 return (
  <motion.div className="language-switcher" initial={{ opacity: 0 }} animate={{ opacity: 1 }}>
   <motion.button
    className="language-toggle"
    onClick={() => setIsOpen(!isOpen)}
    whileHover={{ scale: 1.05 }}
    whileTap={{ scale: 0.95 }}
   >
    <span className="language-icon">{currentLang.icon}</span>
    <span className="language-name">{currentLang.name}</span>
    <motion.div
     animate={{ rotate: isOpen ? 180 : 0 }}
     transition={{ duration: 0.2 }}
    >
     <ChevronDown size={16} />
    </motion.div>
   </motion.button>

   <AnimatePresence>
    {isOpen && (
     <motion.div
      className="language-dropdown"
      initial={{ opacity: 0, y: -10 }}
      animate={{ opacity: 1, y: 0 }}
      exit={{ opacity: 0, y: -10 }}
     >
      {Object.entries(languages).map(([key, lang]) => (
       <motion.button
        key={key}
        className={`language-option ${currentLanguage === key ? 'active' : ''}`}
        onClick={() => {
         onLanguageChange(key);
         setIsOpen(false);
        }
       }
        whileHover={{ x: 5 }}
        whileTap={{ scale: 0.95 }}
       >
        <span className="language-icon">{lang.icon}</span>
        <span className="language-name">{lang.name}</span>
       </motion.button>
      )
       )}
     </motion.div>
```

```
        )
        }
      </AnimatePresence>
    </motion.div>
  );
};
export default LanguageSwitcher;
```

Visualization.js:

```
class Visualizer {
    constructor(containerId, indexContainerId) {
        this.container = document.getElementById(containerId);
        this.indexContainer = document.getElementById(indexContainerId);
        this.currentArray = [];
        this.currentGraph = null;
        this.currentTree = null;
        this.currentStep = null;
        this.visualizationType = 'array';
    }
    initializeArray(array) {
        this.currentArray = [...array]; // Create a copy
        this.visualizationType = 'array';
        this.renderArray();
    }
    initializeGraph(graph) {
        this.currentGraph = graph;
        this.visualizationType = 'graph';
        this.renderGraph();
    // ========== ARRAY VISUALIZATION ==========
    renderArray(highlight = {}) {
        // Clear containers
        this.container.innerHTML = '';
        this.indexContainer.innerHTML = '';
        // Create array elements and indices
        this.currentArray.forEach((value, index) => {
            // Create array element (box with number)
            const element = document.createElement('div');
            element.className = 'array-element';
            element.textContent = value; // This shows the actual number
            element.dataset.index = index;
            element.dataset.value = value;
            // Apply highlighting based on current step
            this.applyArrayHighlight(element, index, highlight);
            this.container.appendChild(element);
            // Create index element
            const indexElement = document.createElement('div');
            indexElement.className = 'index-item';
            indexElement.textContent = index;
            this.indexContainer.appendChild(indexElement);
        }
```

```
      );
    }
    }
     // ========== GRAPH VISUALIZATION ==========
    renderGraph(step = {}) {
        this.container.innerHTML = '';
        this.container.innerHTML = '<div style="text-align: center; padding: 20px; color: #64748b;">Graph Visualization - Coming
Soon</div>';
    }
    // ========== TREE VISUALIZATION ==========
    renderTree(step = {}) {
        this.container.innerHTML = '';
        this.container.innerHTML = '<div style="text-align: center; padding: 20px; color: #64748b;">Tree Visualization - Use BST for
tree operations</div>';
    }
    // ========== MAIN VISUALIZATION METHOD ==========
    visualizeStep(step) {
        this.currentStep = step;
            if (step.array && this.visualizationType === 'array') {
            this.currentArray = [...step.array]; // Update with the actual swapped array
        }
        switch (this.visualizationType) {
            case 'array':
                this.renderArray(step.highlight || {});
                break;
            case 'graph':
                this.renderGraph(step);
                break;
            case 'tree':
                this.renderTree(step);
                break;
        }
    // Method to get current visualization state
    getVisualizationState() {
        return {
            array: [...this.currentArray],
            graph: this.currentGraph ? JSON.parse(JSON.stringify(this.currentGraph)) : null,
            tree: this.currentTree ? [...this.currentTree] : null,
            step: this.currentStep,
            type: this.visualizationType,
            container: this.container,
            indexContainer: this.indexContainer
        };
  }
}
// BST Visualization Controller
class BSTVisualizer {
    constructor(containerId) {
        this.container = document.getElementById(containerId);
        if (!this.container) {
```

```
            console.error('BST container not found:', containerId);
            return;
        }
        this.canvas = document.createElement('canvas');
        this.ctx = this.canvas.getContext('2d');
        this.container.appendChild(this.canvas);
        this.currentBST = { root: null, size: 0 };
        this.highlightedNodes = new Set();
        this.currentOperation = null;
        this.nodePositions = new Map();
        this.setupCanvas();
        this.setupEventListeners();
    }
    setupEventListeners() {
        // Add any event listeners needed for BST interactions
        this.canvas.addEventListener('click', (e) => this.handleCanvasClick(e));
    }
    initializeBST(bst) {
        this.currentBST = bst;
        this.highlightedNodes.clear();
        this.render();
    }
    visualizeStep(step) {
        if (!step) return;
        this.currentOperation = step.type;
        this.highlightedNodes.clear();
        // Update current BST if provided in step
        if (step.bst) {
            this.currentBST = step.bst;
        }
        // Handle different step types
        if (step.current) {
            this.highlightedNodes.add(step.current);
        }
        if (step.inserted) {
            this.highlightedNodes.add(step.inserted);
        }

        if (step.found) {
            this.highlightedNodes.add(step.found);
        }
        if (step.visited) {
            this.highlightedNodes.add(step.visited);
        }
        if (step.comparing) {
            this.highlightedNodes.add(step.comparing);
        }
        this.render();
    }
        calculatePositions() {
```

```
    const nodePositions = new Map();
    const levelHeight = 80;
    if (!this.currentBST || !this.currentBST.root) {
        this.nodePositions = nodePositions;
        this.maxDepth = 0;
        return;
    }
    const calculate = (node, depth, minX, maxX) => {
        if (!node) return null;
        const x = (minX + maxX) / 2;
        const y = depth * levelHeight + 60;
        nodePositions.set(node.value, { x, y, node, depth });
        // Calculate positions for children with proper spacing
        const childWidth = (maxX - minX) / 2;
        if (node.left) {
            calculate(node.left, depth + 1, minX, x - 10);
        }
        if (node.right) {
            calculate(node.right, depth + 1, x + 10, maxX);
        }
    };
    calculate(this.currentBST.root, 0, 50, this.canvas.width - 50);
    this.nodePositions = nodePositions;
}
// ADDED: Draw the complete tree
drawTree() {
    // Draw edges first (so they appear behind nodes)
    this.drawEdges();
    // Then draw nodes (so they appear on top)
    for (const [value, pos] of this.nodePositions) {
        this.drawNode(pos);
    }
}
        // Draw edge to right child
        if (node.right) {
            const rightPos = this.nodePositions.get(node.right.value);
            if (rightPos) {
                this.ctx.beginPath();
                this.ctx.moveTo(x, y);
                this.ctx.lineTo(rightPos.x, rightPos.y);
                this.ctx.stroke();
            }
          }
          }
          }
drawNode(pos) {
    const { x, y, node } = pos;
    const radius = 20;
    // Determine node color based on current operation and highlighting
    let fillColor = '#475569'; // Default gray
```

```
        let borderColor = '#64748b';
        }
        // Draw node circle
        this.ctx.fillStyle = fillColor;
        this.ctx.strokeStyle = borderColor;
        this.ctx.lineWidth = 2;
        this.ctx.beginPath();
        this.ctx.arc(x, y, radius, 0, 2 * Math.PI);
        this.ctx.fill();
        this.ctx.stroke();
        // Draw node value
        this.ctx.fillStyle = '#ffffff';
        this.ctx.font = 'bold 12px Arial';
        this.ctx.textAlign = 'center';
        this.ctx.textBaseline = 'middle';
        this.ctx.fillText(node.value.toString(), x, y);
    }
if (typeof module !== 'undefined' && module.exports) {
    module.exports = { Visualizer, AnimationController, BSTVisualizer };
}
```

VII. ACKNOWLEDGEMENT

It is one of the most efficient tasks in life to choose the appropriate words to express one's gratitude to the beneficiaries. We are very much grateful to God who helped us all the way through the project and how molded us into what we are today.

We are grateful to our beloved Principal Dr. R. RADHAKRISHNAN, M.E., Ph.D., Adhiyamaan College of Engineering (An Autonomous Institution), Hosur for providing the opportunity to do this work in premises.

We acknowledge our heartful gratitude to Dr. G. FATHIMA, M.E., Ph.D., Professor and Head of the Department, Department of Computer Science and Engineering, Adhiyamaan College of Engineering (An Autonomous Institution), Hosur, for her guidance and valuable suggestions and encouragement throughout this project and made us to complete this project successfully.

We are highly indebted to Mrs. A. LAVANYA, M.E., Supervisor, Assistant Professor, Department of Computer Science and Engineering, Adhiyamaan College of Engineering (An Autonomous Institution), Hosur, whose immense support encouragement and valuable guidance were responsible to complete the project successfully.

We also extend our thanks to Project Coordinator and all Staff Members for their support in complete this project successfully.

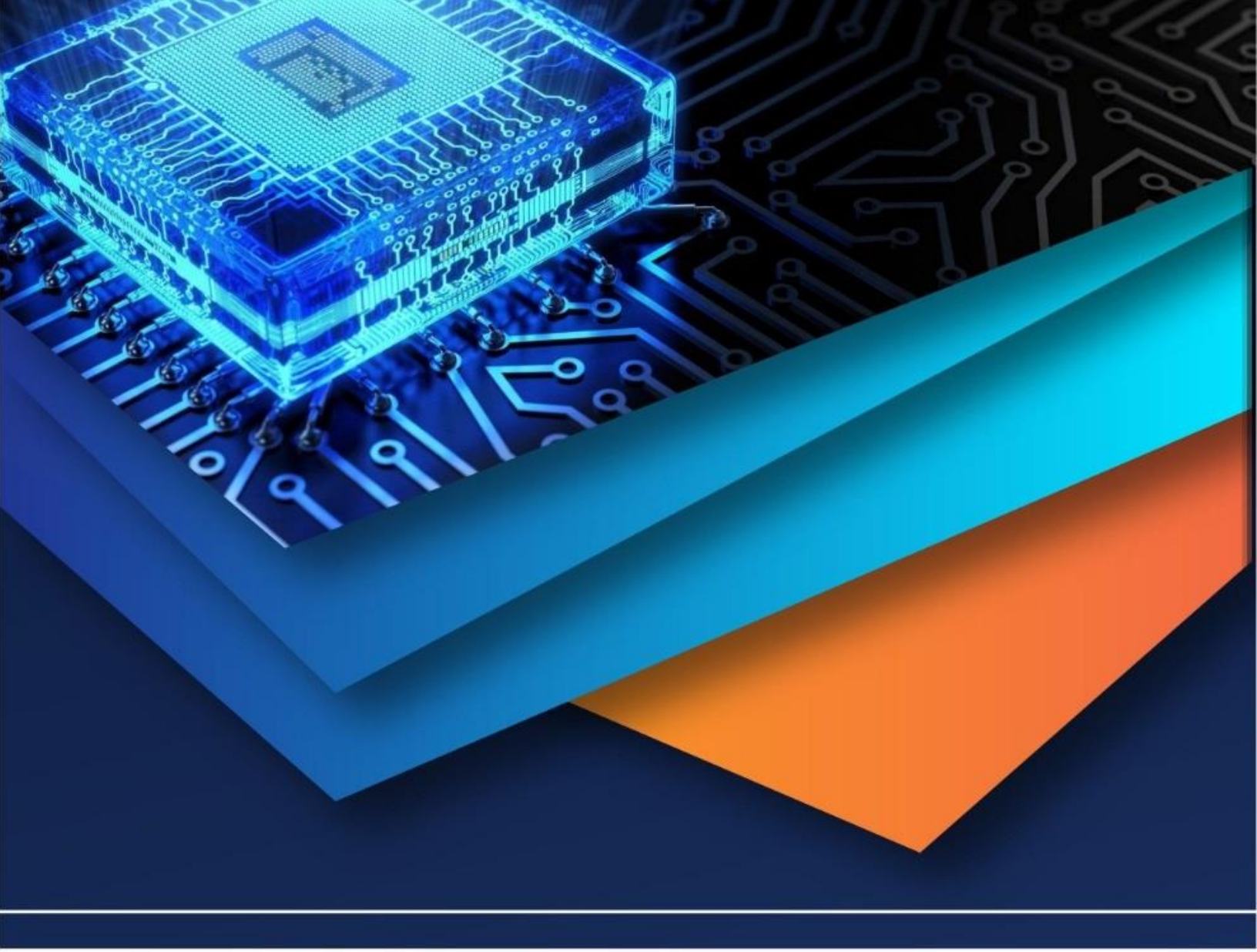
Finally, we would like to thank to our parents, without their motivational and support would not have been possible for us to complete this project successfully.

REFERENCES

[1] S. R. Bhandari et al., "Algorithm Visualization on Data Structures," Journal of Computer Science Education, 2025.
[2] F. Vateha and S. Simoňák, "Comparative Visualization of Algorithms and Data Structures," International Journal of Educational Technology in Computer Science, 2025.
[3] K. R. Sharma and P. Verma, "Immersive Algorithm Animation in Augmented Reality for Data Structure Learning," International Journal of Interactive Multimedia and Artificial Intelligence, 2025.
[4] L. N. Patel, M. Q. Huang & S. Lee, "Visual Trace Mining: An Interactive Tool for Code-and-Data Structure Visualization in Real Time," Journal of Computer Science Education Research, 2025.
[5] J. A. Roberts, E. P. Gómez & R. Singh, "From Pseudocode to Visualization: A Gamified Approach to Algorithm Teaching," Journal of Educational Technology & Society, 2025.
[6] T. W. Kim, A. Brown & V. Singh, "Comparative Study of 3D and 2D Algorithm Visualizations in VR Environments for Undergraduate Learning," Computers & Education, 2025.
[7] H. Z. Ali, S. Munir & D. H. Lee, "Adaptive Visualization Framework for Dynamic Programming Using AI-Driven Interactive Dashboards," IEEE Access, 2025.
[8] S. R. Bhandari, A. Gaikwad, A. Huang, F. Vateha et al., "Modern Perspectives on Algorithm Visualization Techniques," IEEE Transactions on Learning Technologies, 2025.
[9] A. Gaikwad et al., "Visualizing Complexity: Navigating Algorithms with Algorithm Visualizer," ACM SIGCSE Bulletin, 2024.

[10] SIGCSE Demo Team, "Demo 3C: Algovision – An Algorithm Visualization Tool," Proceedings of the ACM Conference on Computer Science Education (SIGCSE), 2024.
[11] G. Kogan, H. Chassidim, and I. Rabaev, "The Efficacy of Animation and Visualization in Teaching Data Structures: A Case Study," Computer Applications in Engineering Education, 2024.
[12] D. H. Lee et al., "dpvis: A Visual and Interactive Learning Tool for Dynamic Programming," International Journal of Computing and Informatics, 2024.
[13] A. Ge Zhang et al., "CFlow: Supporting Semantic Flow Analysis of Students' Code in Programming Problems at Scale," IEEE Transactions on Learning Technologies, 2024.
[14] Y. Ye et al., "Generative AI for Visualization: State of the Art and Future Directions," Artificial Intelligence Review, 2024.
[15] R. Mourato and A. L. Santos, "Educational Program Visualizations Using Synthesised Execution Information," Journal of Computational Education, 2024.
[16] S. Chawla and S. Jindal, "Algorithm Visualization: Bridging the Gap Between Theory and Practice," International Journal of Computer Science Education, 2024.
[17] A. S. M. Venigalla et al., "Using Augmented Reality for Visualizing Control Flows in Programs," IEEE Access, 2023.
[18] M. Paredes-Velasco et al., "Augmented Reality with Algorithm Animation and Their Effect," Journal of Educational Multimedia and Hypermedia, 2023.
[19] B. E.-Okikere Ojugo, C. Ugwu, and L. U. Oghenekaro, "Visualization Tool for Data Structures in Real Time," Journal of Computer Applications Research, 2023.
[20] A. Huang et al., "Visualization of Sorting Algorithms in the Virtual Reality Environment," IEEE Transactions on Visualization and Computer Graphics, 2023.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)