TEZPUR UNIVERSITY

A blue and white logo

Description automatically generated

**Bachelor of Technology**

Computer Science and Engineering

PROJECT REPORT

Topic: Graph editor BASED ON JS

*Submitted By*

**AJAY KUMAR (**CSB22060**)**

*Under the guidance of*

**DR. UTPAL SHARMA**

Professor and Dean, Planning & Development

**DEPARTMENT OF COMPUTER SCIENCE**

**TEZPUR UNIVERSITY, ASSAM**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**TEZPUR UNIVERSITY**

**Tezpur-784028, Assam, India****CERTIFICATE**

This is to certify that the project entitled **“Graph Editor Based on JS”** is a bonafide work carried out by **Ajay Kumar (Enrollment No.: CSB22060)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering**, under the course code **CO317 – Project – I (Using SE Perspective)**, during the **Spring Semester 2025**.

The project has been carried out using a Software Engineering perspective, under the guidance of **Dr. Utpal Sharma**, Professor and Dean, Planning & Development, Department of Computer Science and Engineering, Tezpur University.

|  |  |  |
| --- | --- | --- |
| **Dr. Utpal Sharma** | **Dr. S. Ibotombi Singh** | **Dr. Sarat Saharia** |
| Professor and Dean | Project Co-ordinator | Professor & HOD |
| Dept. of CSE | Dept. of CSE | Dept. of CSE |
| School of Engineering | School of Engineering | School of Engineering |
| Tezpur University | Tezpur University | Tezpur University |
| Signature ......................... | Signature ......................... | Signature ......................... |

**Name of Examiners: Signature with Date:**

1.

2.

3.

Acknowledgement

The completion of this project would not have been possible without the support of several individuals. I would like to express my sincere gratitude to all those who assisted me throughout this work.

Firstly, I extend my heartfelt thanks toDr. Utpal Sharma, my project guide, for his invaluable guidance, support, and expertise. Her encouragement shaped the project in the right direction.

I am also deeply grateful to Dr. Sarat Saharia, Professor and Head of the Department of Computer Science and Engineering, Tezpur University, for his continuous support and motivation, which pushed me to extend my academic boundaries.

My thanks go to Dr. S. Ibotombi Singh, the Project Coordinator, and all the faculty members of the Department of Computer Science and Engineering, Tezpur University, for their constructive feedback and encouragement.

Special thanks to the Tezpur University School of Engineering for providing the essential infrastructure and resources that were crucial to completing the project successfully.

I also acknowledge the contributions of my classmates and friends, whose support and valuable insights helped me through various stages of the project.

Lastly, I am profoundly grateful to my family for their unwavering support and encouragement, which kept me moving forward through the challenges.

I also appreciate anyone else who directly or indirectly contributed to the success of this

Project.

**Ajay Kumar**

**CSB22060**

Abstract

This report provides a comprehensive overview of the Interactive Web-Based Graph Editor, a sophisticated tool designed for the creation, manipulation, and real-time visualization of graph structures and algorithms. Developed using a modern web technology stack including HTML5, CSS3, and JavaScript (ES6), the application facilitates an intuitive user experience for constructing graphs and observing the execution of fundamental algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra’s Algorithm, and an enhanced Bellman-Ford Algorithm. Particular attention was given to robust error handling, especially concerning negative weight cycle detection in the Bellman-Ford implementation. This document elaborates on the project's objectives, system architecture, technology choices with their specific benefits, core features, implementation highlights, achieved outcomes, challenges encountered, and potential avenues for future development, underscoring its significant educational and technical value.

Table of Contents

[**1** **Introduction 7**](#_Toc199088256)

[1.1 Project Profile 7](#_Toc199088257)

[**2** **Initial System Study 8**](#_Toc199088260)

2.1. [Problem Definition 8](#_Toc199088262)

[2.2. Drawbacks of Existing Systems 8](#_Toc199088264)

2.3.The Proposed System  [8](#_Toc199088262)

[2.4. Scope of the Project 9](#_Toc199088264)

[**3** **Feasibility Analysis 10**](#_Toc199088261)

3.1. [Requirements for System Deployment 10](#_Toc199088262)

[3.2. Feasibility Statement **Error! Bookmark not defined.**](#_Toc199088264)

[**4** **Software Requirements Specification (SRS) 11**](#_Toc199088261)

[4.1 Functional Requirements 11](#_Toc199088262)

[4.1.1 Requirement 1: Graph Creation 11](#_Toc199088264)

[4.1.2 Requirement 2: Graph Type 12](#_Toc199088262)

[4.1.3 Requirement 3: Algorithm Visualization 12](#_Toc199088264)

[4.1.4 Requirement 4: Data Management 13](#_Toc199088264)

[4.2 Non-Functional Requirements 13](#_Toc199088262)

[4.3 System Requirements 14](#_Toc199088264)

[**5** **System Analysis 15**](#_Toc199088261)

[5.1 Introduction 15](#_Toc199088262)

[5.2 Data Flow Diagram (DFD) 15](#_Toc199088262)

[5.2.1 Level 0 DFD : 15](#_Toc199088262)

[5.2.2 Level 1 DFD : 16](#_Toc199088262)

[**6** **System Design 17**](#_Toc199088261)

[6.1 Introduction 17](#_Toc199088262)

[6.2 System Architecture 17](#_Toc199088262)

[6.3 Module Design 18](#_Toc199088262)

[6.4 Input/Output Design 19](#_Toc199088262)

[6.5 Algorithm Design 19](#_Toc199088262)

[**7** **System Implementation 22**](#_Toc199088261)

[7.1 Hardware Components 22](#_Toc199088262)

[7.2 Software Environment 22](#_Toc199088262)

[7.3 System Development Platform 23](#_Toc199088262)

[7.4 Project Accomplishment Status 23](#_Toc199088262)

[**8** **System Testing 24**](#_Toc199088261)

[8.1 Test Plan 24](#_Toc199088262)

[8.2 Test Cases 25](#_Toc199088262)

[**9** **Limitations 26**](#_Toc199088261)

[**10** **Future Work 27**](#_Toc199088261)

[**11** **Conclusion 28**](#_Toc199088261)

[11.1. Appendix A 28](#_Toc199088262)

[11.2. Appendix B 29](#_Toc199088262)

Project Report: Interactive Web-Based Graph Editor and Algorithm Visualizer

1. Introduction

**1.1 Project Profile**

**Overview**: This project is a web-based application called **"Graph Editor Using JavaScript"**, developed as a mini project. It allows users to create, edit, and visualize graph data structures. The system is designed using **HTML5**, **JavaScript**, and the **Canvas API**, along with **Bootstrap** for a clean and responsive interface.

With this tool, users can:

* Draw graphs by adding **nodes** (points) and **edges** (connections between points).
* Choose between **directed** and **undirected** graphs.
* Visualize how different **graph algorithms** work such as:
  + **Breadth-First Search (BFS)**
  + **Depth-First Search (DFS)**
  + **Dijkstra’s Algorithm**
  + **Bellman-Ford Algorithm**

The purpose of the project is to help **students**, **teachers**, and **learners** understand and interact with graph algorithms in a visual and intuitive way. The application runs entirely in the browser, so no installation is needed.

2. Initial System Study

**2.1 Problem Definition**

In computer science, understanding graph theory and algorithms can be difficult without a good visual representation. Many learners struggle to understand how algorithms like BFS or Dijkstra’s work because they cannot see how each step is performed.

There is a need for an **easy-to-use** and **interactive** tool that lets users:

* Draw their own custom graphs.
* See how graph algorithms explore the graph.
* Learn through **step-by-step animation** and **visual feedback**.

This project aims to solve that problem.

**2.2 Drawbacks of Existing Systems**

Many current tools or methods used for teaching graphs have several limitations:

* **Manual drawing on paper** is time-consuming and static.
* Some online tools are too **complex**, **non-interactive**, or **not mobile-friendly**.
* Many tools do not show **step-by-step visualizations**, which makes it hard to follow the algorithm.
* Some applications require **installation** or **sign-up**, which can be inconvenient.

These drawbacks limit the ability of students to experiment and understand graph algorithms effectively.

**2.3 The Proposed System**

The proposed solution is a **web application** that allows users to create and manipulate graphs directly in the browser. It is built using client-side technologies, which means:

* No need for a server or database.
* Fast and responsive performance.
* Can be used on any modern device (desktop, tablet, mobile).

The tool includes:

* Graph editing (add/delete nodes and edges).
* Switching between **directed** and **undirected** modes.
* Running and **visualizing graph algorithms** step by step.
* Buttons to **pause**, **resume**, or **step through** the process.
* A panel showing **graph statistics** and **results**.

**2.4 Scope of the Project**

The main focus of this project is to:

* Build a functional **graph editor** that is easy to use.
* Support major graph algorithms with **step-by-step visualization**.
* Allow users to **save** their graphs as files and **load** them later.
* Provide a **responsive design** that works well on both desktop and mobile.
* Maintain smooth performance even with **moderately sized graphs**.

The project is especially suitable for educational purposes, such as:

* Classroom demonstrations.
* Self-learning by students.
* Creating visual aids for presentations or assignments.

In short, this tool bridges the gap between theory and practice in graph algorithms.

technical design, implementation journey, and contributions of this project.

Okay, here's a simpler explanation for the "Feasibility Analysis" section.

3. Feasibility Analysis

This section looks at whether the project is **doable** and **makes sense** to build. It answers questions like: "Can we actually build this with what we have?" and "Is it worth doing?"

**3.1 Requirements for System Deployment**

This part talks about what's needed to actually **use** the "Graph Editor" once it's built. "Deployment" just means making it available for people to use.

* **Hardware (What physical stuff do you need?):**
  + You need a **device** like a computer (desktop or laptop) or a tablet.
  + This device must be able to run a **modern web browser**. Think of popular browsers like Google Chrome, Mozilla Firefox, or Microsoft Edge. The project itself doesn't need a super-powerful computer, just one that can handle basic web Browse and the graphics drawn by the tool.
* **Software (What programs or code are needed?):**
  + The good news is, users don't need to install any special software *other than their web browser*.
  + The tool itself is built using:
    - **HTML:** The basic structure of the webpage.
    - **JavaScript:** The programming language that makes the tool interactive and does all the smart work like drawing the graph and running algorithms.
    - **Bootstrap:** A free toolkit that helps make the website look good and work well on different screen sizes (it handles a lot of the styling).

**3.2 Feasibility Statement**

This is a summary of whether the project is a good idea from different angles.

* **Technical Feasibility (Can we technically build it?):**
  + **Yes, it's achievable.** The project uses standard web technologies that are well-understood and widely available.
  + **JavaScript** is powerful enough to handle the logic for creating graphs and running algorithms.
  + **HTML5 Canvas** is a feature in HTML that is specifically designed for drawing graphics like nodes and edges directly in a web browser.
  + The project also uses a **Web Worker** to run the complex algorithms in the background. This is a smart technical choice because it stops the webpage from freezing or becoming slow while the algorithm is thinking.
* **Economic Feasibility (Will it cost too much?):**
  + **It's low cost.** 💰
  + The tools used to build it are **free**. HTML, JavaScript, and Bootstrap don't cost anything to use. You can write the code in free text editors (like VS Code).
  + It doesn't need expensive servers to run because it's a "client-side" application (it runs in the user's browser).
* **Operational Feasibility (Will it be easy to use and manage?):**
  + **Yes, it should be easy to use.**
  + The design aims to be **user-friendly**, especially for **students** who are learning about graphs, and for **developers** or teachers who might use it as a teaching aid.
  + Because it runs in a web browser, there's no complicated setup or installation process for users. They just open a web link.

4. Software Requirements Specification (SRS)

This section describes the complete software requirements specification (SRS) for the Graph Editor application. It outlines both the functional and non-functional requirements that define how the system should behave, as well as the hardware and software environment necessary for deployment.

**4.1 Functional Requirements**

These are the core features that the system must provide for the user to achieve its intended purpose.

**4.1.1 Requirement 1: Graph Creation**

* **Add Nodes**: Users can click a button to add nodes. Nodes appear on the canvas and can be dragged to different locations.
* **Move Nodes**: Nodes can be repositioned by dragging them with the mouse.
* **Delete Nodes and Edges**: In delete mode, users can click on a node or edge to remove it from the canvas.
* **Draw Edges**: Users can enter "Edge Mode" to connect nodes by clicking on two nodes in sequence and assigning an edge weight.

**4.1.2 Requirement 2: Graph Type**

* **Toggle Graph Direction**: A button allows users to switch between **directed** and **undirected** graph modes.
* **Visual Feedback**: Directed edges are shown with arrowheads; undirected edges are straight lines.
* **Edge Behavior**: Edge connections adapt according to the graph type (e.g., a directed edge from A to B does not imply B to A).

**4.1.3 Requirement 3: Algorithm Visualization**

* **Supported Algorithms**:
  + **Breadth-First Search (BFS)**
  + **Depth-First Search (DFS)**
  + **Dijkstra’s Algorithm**
  + **Bellman-Ford Algorithm**
* **Start Node Selection**: Users choose the starting node through a modal input.
* **Step-by-Step Mode**: Users can run the algorithm step-by-step, observing each action.
* **Animation Control**: Includes Run, Pause, Resume, and Step buttons for algorithm animation.
* **Visualization Feedback**: Highlights visited nodes and traversed edges, showing the current progress.

**4.1.4 Requirement 4: Data Management**

* **Save Graph**: Users can save the current graph as a downloadable .json file containing node and edge data.
* **Load Graph**: Users can upload a .json file to restore a previously saved graph.
* **Undo Action**: An undo feature is available to revert the last action (add/move/delete node or edge).
* **Reset Canvas**: A button clears the entire canvas and resets all graph data.

**4.2 Non-Functional Requirements**

These define how the system performs and behaves beyond its core functionalities.

* **Performance**:
  + The system should support smooth animations and rendering for graphs with up to **50 nodes** and **100 edges**.
  + The canvas must redraw efficiently during dragging and algorithm animation.
* **Usability**:
  + The UI should be simple, clean, and suitable for students and beginner-level developers.
  + Tooltips, icons, and labels should guide users intuitively.
  + Responsive design ensures usability on both desktop and mobile devices.
* **Compatibility**:
  + The application must run seamlessly on modern web browsers such as:
    - **Google Chrome**
    - **Mozilla Firefox**
    - **Microsoft Edge**
  + No additional software or browser plugins should be required.

**4.3 System Requirements**

This section defines the necessary hardware and software components needed to run the application.

* **Hardware Requirements**:
* Device with a minimum of 2GB RAM.
  + Any operating system (Windows, macOS, Linux, Android) that can run a modern browser.
  + Mouse or touch input support.
* **Software Requirements**:
  + **Frontend Technologies**:
    - HTML5 for structure
    - CSS3 and Bootstrap 5.3.2 for styling and layout
    - JavaScript for interactivity and logic
  + **Canvas API** for real-time 2D rendering of the graph.
  + **Web Worker API** to perform background execution of graph algorithms without freezing the UI.
  + **Bootstrap Icons** and **Google Fonts** for enhanced UI experience.
* **Assumptions**:
  + The user has a basic understanding of using a web browser.
  + The application is used in a client-side environment—no server setup is needed.
  + Files are loaded from the user’s local system (via file upload).

5. System Analysis

**5.1 Introduction**

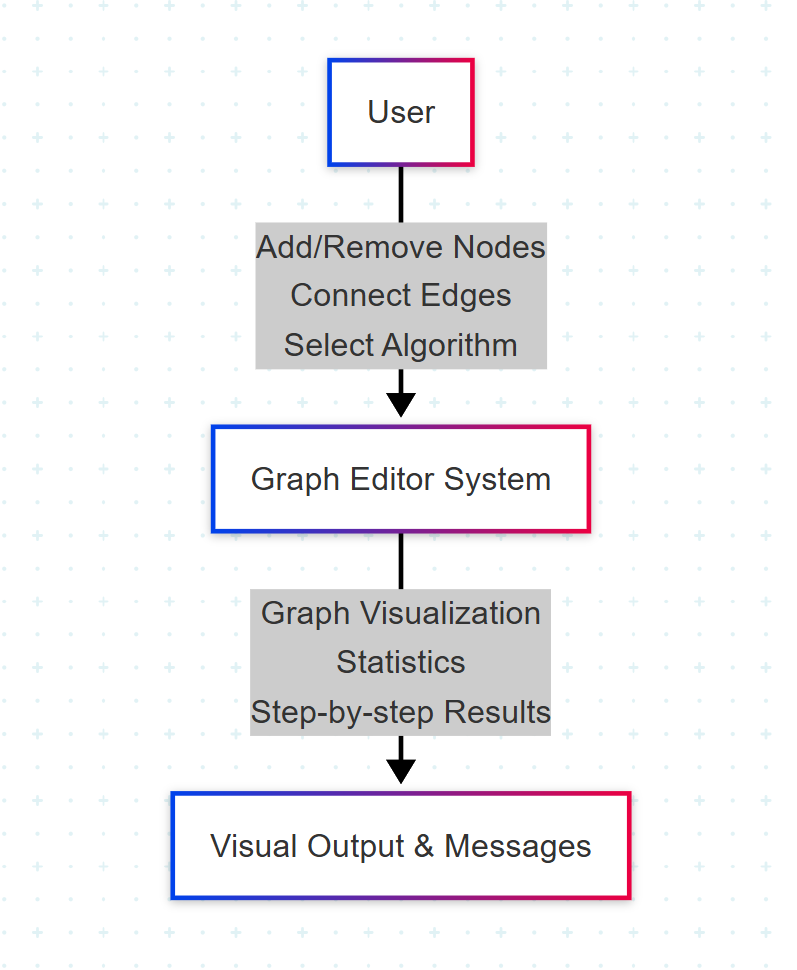
System analysis involves understanding how different components of the Graph Editor work together. The purpose is to break down the internal process of graph editing, user interaction, and how graph algorithms are executed and visualized in the system.

The system is designed to provide a smooth experience where users can interact with the graph visually and run various algorithms step by step. The analysis explains how the user inputs flow through the system and produce the final output.

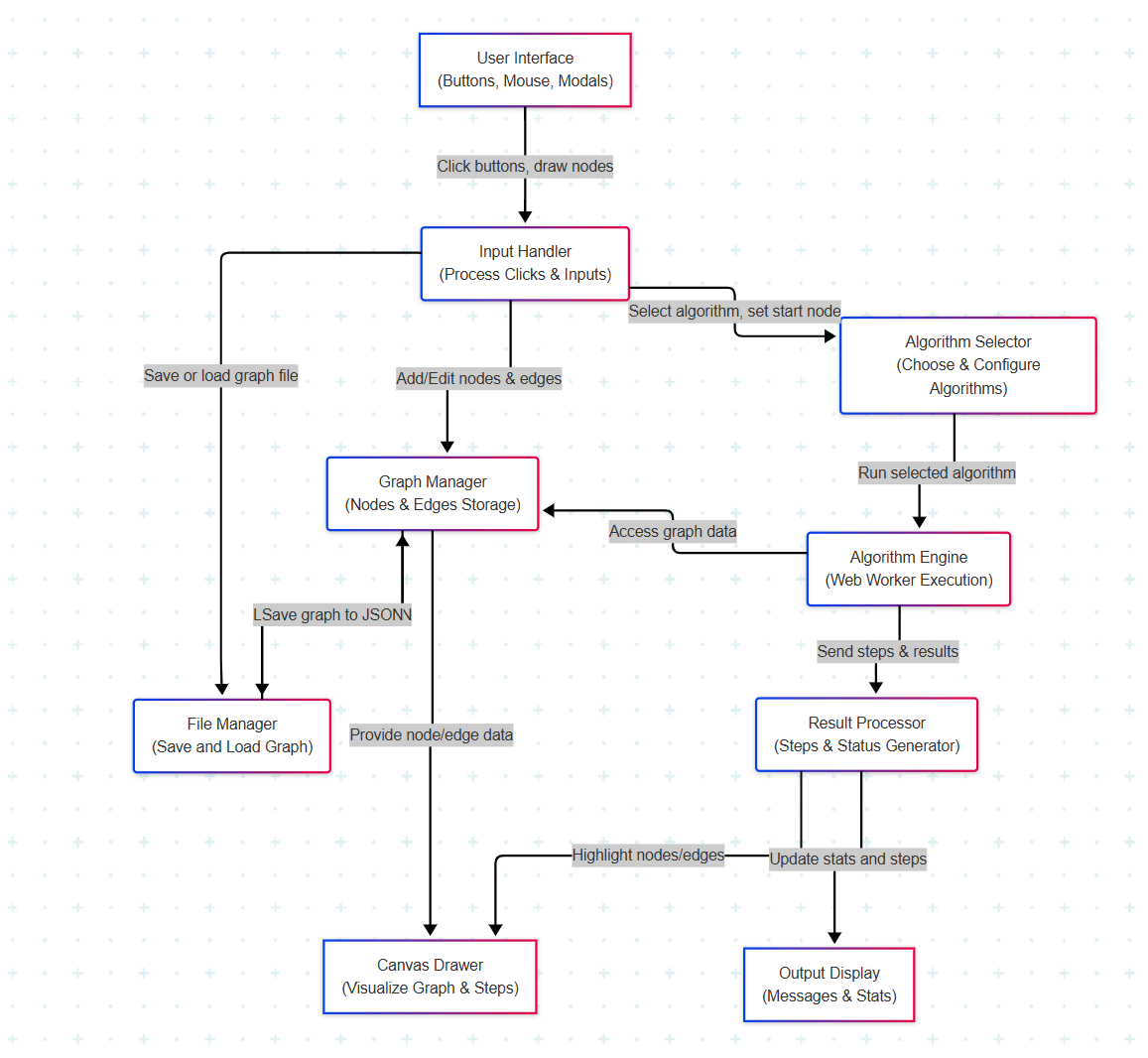
**5.2 Data Flow Diagram (DFD)**

A Data Flow Diagram (DFD) represents how data moves within the system. It helps in understanding the overall architecture and interaction between different components.

**5.2.1 Level 0 DFD – Context Diagram:**

****

**5.2.2 Level-1: Data flow Diagram**

****

**Conceptual Data Flow:**

1. **User Interaction**  
   The user clicks buttons or enters data (like adding nodes, selecting algorithms, or loading a file).
2. **Input Handling**  
   The system captures the user’s actions and sends them to the right part of the program.
3. **Graph Management**  
   The graph structure (nodes and edges) is updated and stored based on user input.
4. **File Operations**  
   Users can save the current graph as a file or load a saved graph using the file manager.
5. **Algorithm Execution**  
   When the user runs an algorithm (like BFS or Dijkstra), it processes the graph in the background.
6. **Canvas Rendering**  
   The graph and algorithm steps are drawn visually on the canvas.
7. **Output Display**  
   Final results, messages, and graph statistics are shown in the output panel.

6. System Design

**6.1 Introduction**

The system is designed to be a simple, modular, and fully client-side web application. Its primary goal is to allow users to easily create and visualize graphs and to understand the behavior of graph algorithms through interactive, step-by-step animations.

The architecture is split into multiple independent modules that handle input events, data structures, algorithm execution, rendering, and output display. This modular approach ensures better readability, maintainability, and scalability.

**6.2 System Architecture**

The system is built entirely on the client side using standard web technologies:

* **HTML5**: Provides the structure of the web page and includes a <canvas> element used for rendering the graph.
* **CSS & Bootstrap**: Used for styling, layout, responsiveness, and consistent UI elements.
* **JavaScript**: Implements all business logic for user interaction, graph operations, rendering, and animations.
* **Web Worker**: Offloads the execution of computationally intensive algorithms (like Dijkstra or Bellman-Ford) to a background thread, ensuring the UI remains responsive.

This architecture allows the application to run without any server, database, or backend infrastructure.

**6.3 Module Design**

The system is divided into the following core modules:

1. **Graph Manager**:
   * Maintains nodes[] and edges[] arrays.
   * Manages graph creation, deletion, and updates.
   * Supports directed and undirected graph toggling.
2. **Input Handler**:
   * Captures mouse events and button interactions.
   * Handles drag, delete, add edge, and undo modes.
   * Triggers modals for edge weight and start node.
3. **Modal Manager**:
   * Displays modal popups for edge weight input and algorithm start node.
   * Validates user inputs and updates graph state.
4. **Algorithm Engine (Web Worker)**:
   * Executes BFS, DFS, Dijkstra’s, and Bellman-Ford.
   * Returns step-by-step visual instructions to the main thread.
5. **Step Generator & Animation**:
   * Generates animation steps (highlight, visit, backtrack, etc.).
   * Supports pause/resume/step-through functionality.
6. **Canvas Renderer**:
   * Uses drawGraph() to visually render nodes and edges.
   * Handles gradients, highlights, self-loops, and arrowheads.
7. **File Manager**:
   * Handles saving and loading graphs via JSON files.
   * Parses uploaded data and restores complete graph state.
8. **Output Manager**:
   * Displays graph stats (nodes, edges, components, etc.).
   * Shows alerts, warnings, and algorithm results.

**6.4 Input/Output Design**

**Input**

* **Mouse Clicks**: Used to select, drag, or delete nodes; connect edges.
* **Buttons**: Used to toggle modes (edge/delete), reset canvas, undo actions.
* **Modals**: Used to input edge weights and specify the start node for algorithms.
* **File Upload**: Used to load .json graph files into the application.

**Output**

* **Canvas Display**: Shows graph structure, edges, node positions, highlights, and animations.
* **Output Panel**:
  + Shows alerts, errors, success messages.
  + Displays statistics such as number of nodes, edges, graph type, self-loops, and connected components.
  + Shows results of algorithm execution (like shortest paths).

**6.5 Algorithm Design**

The system supports four major graph algorithms. Each is implemented in a Web Worker to avoid blocking the main thread and ensure a smooth user experience. The worker emits animation steps, which are visualized using the canvas.

1. **Breadth-First Search (BFS)**
   * **Traversal Type**: Level-order traversal using a **queue**.
   * **Purpose**: Find the shortest path (in terms of edges) from a source node to all reachable nodes in an unweighted graph.
   * **Steps**:
     + Start from the given source node.
     + Visit neighbors layer by layer.
     + Mark visited nodes and edges.
     + Record the sequence of visits and animate each step.
   * **Edge Cases**:
     + Handles disconnected components gracefully.
     + Skips already visited nodes.
2. **Depth-First Search (DFS)**
   * **Traversal Type**: Recursive or stack-based traversal.
   * **Purpose**: Explore all paths from the source node by going as deep as possible before backtracking.
   * **Steps**:
     + Start from the selected node.
     + Visit the deepest unvisited node first.
     + Backtrack when no more options are available.
     + Animate the progress, including backtracking steps.
   * **Edge Cases**:
     + Recursively avoids cycles.
     + Displays visual trace of recursive stack (color changes or labels).
3. **Dijkstra’s Algorithm**
   * **Goal**: Find the **shortest path** from a source node to all other nodes in a **graph with non-negative weights**.
   * **Data Structures Used**: Priority queue (min-heap), distance table, parent map.
   * **Steps**:
     + Initialize distances to all nodes as infinity, source as 0.
     + Repeatedly select the node with the smallest tentative distance.
     + Update distances of its neighbors if a shorter path is found.
     + Animate node selection and edge relaxation visually.
   * **Edge Cases**:
     + Ignores edges with negative weights.
     + Can handle disconnected graphs by skipping unreachable nodes.
4. **Bellman-Ford Algorithm**
   * **Goal**: Find the shortest path in a graph with **negative weights** and detect **negative-weight cycles**.
   * **Data Structures Used**: Distance table, edge list.
   * **Steps**:
     + Initialize distances to infinity, source to 0.
     + Repeat edge relaxation for all edges (V-1) times.
     + Check for negative-weight cycles in a final iteration.
     + Each edge relaxation is shown visually, highlighting the change.
   * **Edge Cases**:
     + Displays a warning message when a negative-weight cycle is detected.
     + Skips unreachable nodes during traversal.

**Common Features Across All Algorithms**

* **Visualization**: Each step (visit, relax, backtrack) is shown using color changes and arrows.
* **Animation Controls**: The user can control playback with Run, Pause, Resume, and Step buttons.
* **Start Node Input**: Taken via modal input before algorithm execution.
* **Responsive Behavior**: Algorithms continue to work properly on graphs loaded from files.

7. System Implementation

**7.1 Hardware Components**

This project is a browser-based tool that runs on client-side hardware without the need for any server infrastructure. However, it does have some limitations in terms of device support.

**Recommended Hardware**

* **Desktop or Laptop Computer** with mouse support is highly recommended for smooth interaction.
* **Minimum RAM**: 4 GB (to ensure smooth canvas rendering and JavaScript execution).
* **Display**: Minimum 1366×768 resolution recommended for full toolbar visibility and better canvas interaction.

**Not Supported**

* **Mobile Devices (Smartphones/Tablets)**:  
  The Graph Editor interface is **not fully supported on mobile devices**. This is due to:
  + Small screen sizes making it difficult to interact with canvas elements precisely.
  + Lack of full support for dragging and precise edge creation using touch input.
  + Mobile event handling and gesture recognition being different from desktop mouse events, which are used throughout the application.

Thus, the project is best experienced on a desktop environment with a full browser.

**7.2 Software Environment**

The Graph Editor was built entirely using **open-source, client-side web technologies**, making it highly portable and cost-effective.

| **Component** | **Description** |
| --- | --- |
| **HTML5** | Used to structure the webpage and embed the canvas element for drawing graphs. |
| **CSS3 + Bootstrap 5.3.2** | Provides modern, responsive styling for UI elements like toolbars, buttons, modals, and layout structure. |
| **JavaScript** | Implements logic for graph construction, manipulation, animations, and event handling. |
| **Canvas API** | Renders graph nodes and edges visually in real time. |
| **Web Worker API** | Runs graph algorithms (BFS, DFS, Dijkstra, Bellman-Ford) asynchronously in a background thread to prevent the UI from freezing. |
| **Bootstrap Icons** | Provides vector icons for toolbar buttons. |
| **Google Fonts (Poppins)** | Used for clean and modern typography across the interface. |

No frameworks (e.g., React, Angular) were used. The project runs purely in the browser without backend services.

**7.3 System Development Platform**

**Development Tools Used**

* **Text Editor**: Visual Studio Code (VS Code) – used for writing HTML, CSS, and JavaScript with extensions for live preview.
* **Browsers Used for Testing**:
  + Google Chrome (latest version)
  + Mozilla Firefox
* **Operating System**: Windows 11 (development), also tested on Ubuntu 22.04.

**Project Structure**

* index.html: Defines the user interface and includes styles and buttons for toolbar actions.
* script.js: Contains all JavaScript logic for canvas interaction, graph management, event handling, algorithm animations, and undo functionality.

**7.4 Project Accomplishment Status**

The system has been successfully developed with all core functionalities working as intended on desktop browsers:

✅ **Graph Creation and Management**:

* Add, move, delete nodes and edges.
* Toggle between directed and undirected graphs.
* Real-time edge weight input and start node selection using modals.

✅ **Algorithm Support and Visualization**:

* BFS (Breadth-First Search)
* DFS (Depth-First Search)
* Dijkstra’s Shortest Path Algorithm
* Bellman-Ford Algorithm (including negative edge weights)
* Step-by-step execution with animation, visual highlights, pause/resume, and speed control.

✅ **Graph Data Handling**:

* Save graph as a .json file.
* Load graphs from existing .json files.
* Undo last action.
* Reset the entire canvas and start fresh.

✅ **Interface and UX**:

* Fully responsive layout on desktops.
* Dark-themed UI with tooltips, animations, and smooth transitions.

⚠️ **Limitations**:

* Mobile support is **not functional**.
* Touch gestures and canvas event handling are optimized for mouse input only.

8. System Testing

**8.1 Test Plan**

To ensure reliability and correctness, the system was tested through a series of manual test cases. The objective was to:

* Verify all functionalities.
* Check behavior under different conditions.
* Validate user interactions, including algorithm execution and undo/reset actions.
* Observe performance under higher graph complexity.

**8.2 Test Cases and Results**

| **Test No.** | **Description** | **Expected Result** | **Status** |
| --- | --- | --- | --- |
| **TC1** | Add 10 nodes, 15 edges | Nodes and edges are placed correctly and drawn without overlapping | ✅ Passed |
| **TC2** | Drag a node across the canvas | Node should move smoothly, and all connected edges update automatically | ✅ Passed |
| **TC3** | Delete a node with connected edges | Node and its edges should be removed; stats should update | ✅ Passed |
| **TC4** | Toggle between directed/undirected modes | Arrows on edges should appear/disappear correctly | ✅ Passed |
| **TC5** | Run BFS algorithm from selected start node | Traversal path is shown step-by-step with animation | ✅ Passed |
| **TC6** | Run DFS algorithm | DFS traversal is animated correctly with recursion path visualization | ✅ Passed |
| **TC7** | Run Dijkstra with positive weights | Correct shortest path is displayed; final distances are shown | ✅ Passed |
| **TC8** | Run Dijkstra with negative weights | Displays an error message; algorithm does not execute | ✅ Passed |
| **TC9** | Run Bellman-Ford with negative weights | Executes properly; shortest path shown and negative weights handled | ✅ Passed |
| **TC10** | Save a graph and reload it using the load function | Graph structure (nodes, positions, weights) restored exactly | ✅ Passed |
| **TC11** | Press Undo after deleting a node | Last action is reversed and node is restored | ✅ Passed |
| **TC12** | Reset graph after building a large graph | Canvas clears all elements, resets to initial state | ✅ Passed |
|  |  |  |  |

9. Limitations

Despite successful implementation of core features, the Graph Editor has several limitations that can be addressed in future versions:

**9.1 Performance Constraints**

* The application is built using the **HTML5 Canvas API**, which is powerful but becomes performance-heavy as the number of nodes and edges increases.
* When the graph grows beyond **50 nodes** or **100 edges**, users may experience noticeable **lag in rendering and interaction**, especially on lower-end systems.
* This is due to canvas being a pixel-based rendering system, which redraws the entire scene on every change.

**9.2 Feature Limitations**

* **No Zoom/Pan Support**: Users cannot zoom into large graphs or pan across the canvas, which limits usability for complex structures.
* **No Keyboard Navigation**: Currently, all actions rely on mouse input. There are no shortcut keys or accessibility features like keyboard tabbing.
* **Limited Algorithm Support**: The system supports only four graph algorithms (BFS, DFS, Dijkstra, Bellman-Ford). Other useful algorithms like A\*, Kruskal’s, and Prim’s are not included.
* **No Graph Editing on Mobile**: Although the UI is responsive, **mobile devices do not support interactive canvas-based editing**, as dragging and precise edge creation are unreliable via touch.

**9.3 Error Handling**

* **Validation is minimal** for certain inputs, such as edge weights or invalid start node IDs.
* **Invalid edge cases** (e.g., self-loops with undefined behavior, circular dependencies in Bellman-Ford) may not always be caught gracefully.
* **Visual glitches** or missed animations may occur if incorrect values are entered or if the system is stressed with rapid actions.

10. Future Work

Several improvements and extensions can be made to enhance both functionality and user experience in future iterations of the Graph Editor:

**10.1 Algorithm Extensions**

* Add advanced algorithms such as:
  + **A\*** for heuristic-based pathfinding.
  + **Kruskal’s** and **Prim’s** algorithms for Minimum Spanning Tree (MST).
  + **Topological Sort** for Directed Acyclic Graphs.
  + **Floyd-Warshall** for all-pairs shortest paths.

**10.2 Performance Optimization**

* Consider replacing or augmenting the Canvas rendering with **WebGL**, which provides hardware-accelerated graphics and can handle larger graph visualizations more smoothly.
* Optimize the **drawing cycle** by updating only parts of the canvas instead of full re-rendering.

**10.3 UI and Usability Enhancements**

* Implement **Zoom and Pan** functionality for better navigation in large graphs.
* Add **Dark Mode** toggle and additional accessibility options like:
  + Keyboard shortcuts
  + ARIA labels for screen readers
  + High-contrast UI modes

**10.4 Robust Error Handling**

* Introduce stricter **input validation** for edge weights, start node IDs, and file imports.
* Improve feedback messages for invalid operations and unexpected behaviors.
* Handle complex edge cases gracefully (e.g., cycles in Bellman-Ford, disconnected components).

11. Conclusion

This mini project successfully demonstrates a working **Graph Editor** developed using **HTML5 Canvas and JavaScript**, designed primarily for educational use.

**Key Achievements**

* Developed an intuitive **web-based interface** for creating and editing graphs.
* Implemented interactive **algorithm visualizations** for BFS, DFS, Dijkstra, and Bellman-Ford.
* Provided essential utilities like **save/load**, **undo**, and **graph statistics**.
* Focused on **user experience** with responsive design (desktop only), animated transitions, and tooltip guidance.

**Educational Impact**

The project is a **valuable tool for students** to:

* Learn how graph algorithms operate step-by-step.
* Experiment visually with different graph structures.
* Develop deeper understanding of data structures and pathfinding logic.

It meets the academic goals of a BTech mini project while offering room for further development.

**Appendix A**

**Input Forms in the System**

* **Edge Weight Input Modal**: Allows user to enter weight value when connecting two nodes.
* **Start Node Input Modal**: Enables user to define the starting point for algorithm execution.

**Output Formats in the System**

* **Canvas Output**:
  + Visual representation of graph with animated edges and highlighted nodes.
  + Dynamic rendering of traversal paths and algorithm results.
* **Output Panel**:
  + Displays statistics like node/edge count, graph type, connected components.
  + Shows alerts and messages related to actions and errors.

**Appendix B – Bibliography**

1. **Bootstrap 5.3.2 Documentation**  
   <https://getbootstrap.com/docs/5.3/>
2. **HTML5 Canvas Tutorials**  
   <https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial>
3. **Graph Theory Reference**
   * Data Structures and Algorithms in JavaScript – Aditya Bhargava
   * Graph Theory by Reinhard Diestel (conceptual reference)
4. **JavaScript Guides**  
   <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>
5. **Web Workers API Documentation**  
   <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API>