

Nile University

Graph Algorithm Visualizer

An Interactive Web-Based Tool for Visualizing Graph
Traversal and Shortest Path Algorithms

CSCI208: Design and Analysis of Algorithms

Supervised by:

Dr. Shereen Aly

Eng. Ganat Elsayed

Eng. Bassant Ahmed Farouk

Prepared by:

Mohamed Mohsen (221001411)

Farah Khaled (221001643)

Omar Ahmed (221001335)

December 2025

Abstract

Graph Algorithm Visualizer is an interactive web-based educational tool designed to help students and developers understand fundamental graph algorithms through real-time visualization. The application provides an intuitive interface for creating, editing, and manipulating graphs while observing step-by-step execution of various traversal and shortest path algorithms.

This project implements five key algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's Algorithm, Bellman-Ford Algorithm, and A* Search Algorithm. Each algorithm is visualized with color-coded nodes and edges, animation controls, and detailed step logs that explain the algorithm's decision-making process at each iteration.

The application is built using modern web technologies including React, TypeScript, Vite, and Tailwind CSS, ensuring a responsive, performant, and maintainable codebase. The interactive canvas supports pan and zoom functionality, allowing users to work with graphs of varying sizes and complexity.

Keywords: Graph Algorithms, Visualization, BFS, DFS, Dijkstra, Bellman-Ford, A*, React, TypeScript, Web Application

Contents

Abstract	1
1 Introduction	8
1.1 Background and Motivation	8
1.2 Problem Statement	8
1.3 Objectives	9
1.4 Scope	9
2 Theoretical Background	10
2.1 Graph Fundamentals	10
2.1.1 Definition	10
2.1.2 Types of Graphs	10
2.1.3 Graph Representations	11
2.2 Breadth-First Search (BFS)	11
2.2.1 Algorithm Description	11
2.2.2 Pseudocode	12
2.2.3 Complexity Analysis	12
2.2.4 Applications	12
2.3 Depth-First Search (DFS)	13
2.3.1 Algorithm Description	13
2.3.2 Pseudocode	13
2.3.3 Complexity Analysis	13
2.3.4 Applications	13
2.4 Dijkstra's Algorithm	14
2.4.1 Algorithm Description	14
2.4.2 Pseudocode	14
2.4.3 Complexity Analysis	14

2.4.4	Limitations	15
2.5	Bellman-Ford Algorithm	15
2.5.1	Algorithm Description	15
2.5.2	Pseudocode	15
2.5.3	Complexity Analysis	15
2.5.4	Advantages over Dijkstra	16
2.6	A* Search Algorithm	16
2.6.1	Algorithm Description	16
2.6.2	Heuristics	16
2.6.3	Pseudocode	17
2.6.4	Complexity Analysis	17
3	System Design	18
3.1	Architecture Overview	18
3.2	Technology Stack	18
3.3	Component Structure	19
3.3.1	Component Hierarchy	19
3.3.2	Component Descriptions	19
3.4	Data Structures	19
3.4.1	Graph Representation	19
3.4.2	Algorithm Step	20
3.5	State Management	20
4	Implementation	22
4.1	Project Structure	22
4.2	Key Features Implementation	23
4.2.1	Interactive Canvas	23
4.2.2	Algorithm Visualization	24
4.2.3	Visual State Colors	24
4.3	Algorithm Implementations	24
4.3.1	BFS Implementation	24
4.3.2	Dijkstra Implementation	25
4.3.3	DFS Implementation	26

4.3.4	Bellman-Ford Implementation	28
4.3.5	A* Implementation	31
4.4	Deployment Configuration	34
4.4.1	GitHub Actions Workflow	34
5	User Guide	36
5.1	Getting Started	36
5.1.1	Accessing the Application	36
5.1.2	Interface Overview	36
5.1.3	Application Interface	37
5.2	Creating a Graph	37
5.2.1	Adding Nodes	37
5.2.2	Adding Edges	37
5.2.3	Graph Settings	37
5.3	Running Algorithms	38
5.3.1	Setup	38
5.3.2	Playback Controls	38
5.4	Keyboard Shortcuts	38
5.5	Pan and Zoom	39
6	Testing and Validation	40
6.1	Test Cases	40
6.1.1	Graph Operations	40
6.1.2	Algorithm Tests	40
6.2	Browser Compatibility	41
7	Conclusion	42
7.1	Summary	42
7.2	Key Achievements	42
7.3	Future Enhancements	43
7.4	Lessons Learned	43
A	Source Code Repository	45
A.1	Installation Instructions	45

B Team Contributions	46
----------------------	----

List of Figures

5.1 Graph Algorithm Visualizer main interface showing the interactive canvas, toolbar, and algorithm panel	37
--	----

List of Tables

2.1	Types of Graphs	10
2.2	BFS Complexity	12
2.3	DFS Complexity	13
2.4	Dijkstra's Algorithm Complexity	14
2.5	Bellman-Ford Complexity	15
2.6	A* Algorithm Complexity	17
3.1	Technology Stack	18
3.2	Component Descriptions	19
4.1	Node Color States	24
5.1	Keyboard Shortcuts	38
6.1	Graph Operation Tests	40
6.2	Algorithm Test Results	40
6.3	Browser Compatibility	41
B.1	Team Member Contributions	46

Chapter 1

Introduction

1.1 Background and Motivation

Graph theory is a fundamental area of computer science with applications ranging from social networks and routing protocols to recommendation systems and artificial intelligence. Understanding graph algorithms is crucial for any computer science student, yet the abstract nature of these algorithms often makes them challenging to grasp through textbook descriptions alone.

Traditional methods of teaching graph algorithms rely heavily on static diagrams and pseudocode, which fail to capture the dynamic nature of algorithm execution. Students often struggle to visualize how data structures change over time, how decisions are made at each step, and how different algorithms compare in their approach to solving similar problems.

1.2 Problem Statement

The primary challenges in learning graph algorithms include:

- **Abstract Representation:** Graph algorithms operate on abstract data structures that are difficult to visualize mentally.
- **Dynamic Nature:** Understanding how the state changes at each step requires significant cognitive effort.

- **Comparative Analysis:** Comparing the behavior of different algorithms on the same graph is challenging without proper tools.
- **Lack of Interactivity:** Most educational resources provide only passive learning experiences.

1.3 Objectives

The Graph Algorithm Visualizer project aims to:

1. Provide an interactive platform for creating and manipulating graphs
2. Visualize the step-by-step execution of fundamental graph algorithms
3. Enable comparison between different algorithms on the same graph
4. Offer detailed explanations and complexity analysis for each algorithm
5. Create an engaging and intuitive user experience

1.4 Scope

This project covers the following algorithms:

- **Graph Traversal:** BFS and DFS
- **Shortest Path:** Dijkstra's Algorithm, Bellman-Ford Algorithm, and A* Search

The application supports both directed and undirected graphs, as well as weighted and unweighted edges.

Chapter 2

Theoretical Background

2.1 Graph Fundamentals

2.1.1 Definition

A graph $G = (V, E)$ consists of:

- V : A finite set of vertices (nodes)
- E : A set of edges connecting pairs of vertices

2.1.2 Types of Graphs

Table 2.1: Types of Graphs

Type	Description
Directed Graph	Edges have a direction from source to target
Undirected Graph	Edges have no direction (bidirectional)
Weighted Graph	Edges have associated numerical weights
Unweighted Graph	All edges have equal weight (typically 1)
Connected Graph	There exists a path between every pair of vertices
Cyclic Graph	Contains at least one cycle
Acyclic Graph	Contains no cycles

2.1.3 Graph Representations

Adjacency Matrix

An adjacency matrix A is a $|V| \times |V|$ matrix where:

$$A[i][j] = \begin{cases} w_{ij} & \text{if edge exists from } v_i \text{ to } v_j \\ 0 \text{ or } \infty & \text{otherwise} \end{cases}$$

Space Complexity: $O(V^2)$

Adjacency List

An adjacency list stores, for each vertex, a list of its adjacent vertices.

Space Complexity: $O(V + E)$

2.2 Breadth-First Search (BFS)

2.2.1 Algorithm Description

BFS explores a graph level by level, visiting all neighbors of a node before moving to their neighbors. It uses a queue data structure to maintain the order of exploration.

2.2.2 Pseudocode

Algorithm 1 Breadth-First Search

```

1: procedure BFS( $G, s$ )
2:   Create queue  $Q$ 
3:   Mark  $s$  as visited
4:   Enqueue  $s$  into  $Q$ 
5:   while  $Q$  is not empty do
6:      $u \leftarrow$  Dequeue from  $Q$ 
7:     for each neighbor  $v$  of  $u$  do
8:       if  $v$  is not visited then
9:         Mark  $v$  as visited
10:        Enqueue  $v$  into  $Q$ 
11:       end if
12:     end for
13:   end while
14: end procedure
  
```

2.2.3 Complexity Analysis

Table 2.2: BFS Complexity

Metric	Complexity
Time Complexity	$O(V + E)$
Space Complexity	$O(V)$

2.2.4 Applications

- Finding shortest path in unweighted graphs
- Level-order traversal
- Finding connected components
- Web crawling
- Social network analysis

2.3 Depth-First Search (DFS)

2.3.1 Algorithm Description

DFS explores a graph by going as deep as possible along each branch before backtracking. It uses a stack (or recursion) to maintain the order of exploration.

2.3.2 Pseudocode

Algorithm 2 Depth-First Search

```

1: procedure DFS( $G, s$ )
2:   Create stack  $S$ 
3:   Push  $s$  onto  $S$ 
4:   while  $S$  is not empty do
5:      $u \leftarrow$  Pop from  $S$ 
6:     if  $u$  is not visited then
7:       Mark  $u$  as visited
8:       for each neighbor  $v$  of  $u$  do
9:         Push  $v$  onto  $S$ 
10:      end for
11:    end if
12:   end while
13: end procedure
```

2.3.3 Complexity Analysis

Table 2.3: DFS Complexity

Metric	Complexity
Time Complexity	$O(V + E)$
Space Complexity	$O(V)$

2.3.4 Applications

- Topological sorting
- Detecting cycles
- Finding strongly connected components
- Maze solving

- Path finding

2.4 Dijkstra's Algorithm

2.4.1 Algorithm Description

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights. It uses a greedy approach, always selecting the vertex with the minimum distance.

2.4.2 Pseudocode

Algorithm 3 Dijkstra's Algorithm

```

1: procedure DIJKSTRA( $G, s$ )
2:   Initialize  $dist[v] \leftarrow \infty$  for all  $v \in V$ 
3:    $dist[s] \leftarrow 0$ 
4:   Create priority queue  $Q$  with all vertices
5:   while  $Q$  is not empty do
6:      $u \leftarrow$  Extract-Min from  $Q$ 
7:     for each neighbor  $v$  of  $u$  do
8:       if  $dist[u] + w(u, v) < dist[v]$  then
9:          $dist[v] \leftarrow dist[u] + w(u, v)$ 
10:         $prev[v] \leftarrow u$ 
11:        Decrease-Key( $Q, v, dist[v]$ )
12:       end if
13:     end for
14:   end while
15: end procedure

```

2.4.3 Complexity Analysis

Table 2.4: Dijkstra's Algorithm Complexity

Implementation	Time Complexity	Space Complexity
Array	$O(V^2)$	$O(V)$
Binary Heap	$O((V + E) \log V)$	$O(V)$
Fibonacci Heap	$O(E + V \log V)$	$O(V)$

2.4.4 Limitations

- Does not work with negative edge weights
- May not find shortest path in graphs with negative cycles

2.5 Bellman-Ford Algorithm

2.5.1 Algorithm Description

The Bellman-Ford algorithm finds the shortest paths from a source vertex to all other vertices, even when edge weights are negative. It can also detect negative cycles.

2.5.2 Pseudocode

Algorithm 4 Bellman-Ford Algorithm

```

1: procedure BELLMANFORD( $G, s$ )
2:   Initialize  $dist[v] \leftarrow \infty$  for all  $v \in V$ 
3:    $dist[s] \leftarrow 0$ 
4:   for  $i = 1$  to  $|V| - 1$  do
5:     for each edge  $(u, v) \in E$  do
6:       if  $dist[u] + w(u, v) < dist[v]$  then
7:          $dist[v] \leftarrow dist[u] + w(u, v)$ 
8:          $prev[v] \leftarrow u$ 
9:       end if
10:    end for
11:   end for
12:   for each edge  $(u, v) \in E$  do
13:     if  $dist[u] + w(u, v) < dist[v]$  then
14:       report "Negative cycle detected"
15:     end if
16:   end for
17: end procedure
```

2.5.3 Complexity Analysis

Table 2.5: Bellman-Ford Complexity

Metric	Complexity
Time Complexity	$O(V \cdot E)$
Space Complexity	$O(V)$

2.5.4 Advantages over Dijkstra

- Handles negative edge weights
- Can detect negative cycles
- Simpler implementation

2.6 A* Search Algorithm

2.6.1 Algorithm Description

A* is an informed search algorithm that uses heuristics to guide its search. It combines the actual cost from the start (g) with an estimated cost to the goal (h) to prioritize nodes.

The evaluation function is:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$: Actual cost from start to node n
- $h(n)$: Heuristic estimate from node n to goal
- $f(n)$: Total estimated cost through node n

2.6.2 Heuristics

For A* to find the optimal path, the heuristic must be **admissible** (never overestimate) and ideally **consistent**.

Common heuristics:

- **Euclidean Distance:** $h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2}$
- **Manhattan Distance:** $h(n) = |x_n - x_g| + |y_n - y_g|$

2.6.3 Pseudocode

Algorithm 5 A* Search Algorithm

```

1: procedure ASTAR( $G, s, goal$ )
2:   Initialize open set with  $s$ 
3:    $g[s] \leftarrow 0$ 
4:    $f[s] \leftarrow h(s, goal)$ 
5:   while open set is not empty do
6:      $u \leftarrow$  node in open set with lowest  $f$  value
7:     if  $u = goal$  then
8:       return reconstruct_path( $u$ )
9:     end if
10:    Remove  $u$  from open set
11:    for each neighbor  $v$  of  $u$  do
12:       $tentative\_g \leftarrow g[u] + w(u, v)$ 
13:      if  $tentative\_g < g[v]$  then
14:         $prev[v] \leftarrow u$ 
15:         $g[v] \leftarrow tentative\_g$ 
16:         $f[v] \leftarrow g[v] + h(v, goal)$ 
17:        if  $v$  not in open set then
18:          Add  $v$  to open set
19:        end if
20:      end if
21:    end for
22:  end while
23: end procedure

```

2.6.4 Complexity Analysis

Table 2.6: A* Algorithm Complexity

Metric	Complexity
Time Complexity	$O(E)$ (with perfect heuristic) to $O(b^d)$
Space Complexity	$O(V)$

where b is the branching factor and d is the depth of the solution.

Chapter 3

System Design

3.1 Architecture Overview

The Graph Algorithm Visualizer follows a component-based architecture using React.

The application is structured into distinct layers:

1. **Presentation Layer:** React components for UI rendering
2. **Business Logic Layer:** Algorithm implementations and graph utilities
3. **Data Layer:** State management using React hooks

3.2 Technology Stack

Table 3.1: Technology Stack

Category	Technology	Purpose
Frontend Framework	React 18.2	Component-based UI development
Language	TypeScript 5.3	Type-safe JavaScript
Build Tool	Vite 5.0	Fast development server and bundler
Styling	Tailwind CSS 3.4	Utility-first CSS framework
Canvas	HTML5 Canvas	Graph rendering and interaction
Deployment	GitHub Pages	Static site hosting

3.3 Component Structure

3.3.1 Component Hierarchy

```
App
  Header
  Toolbar
  GraphCanvas
  AlgorithmPanel
  MatrixDisplay
  StepLog
  About
```

3.3.2 Component Descriptions

Table 3.2: Component Descriptions

Component	Responsibility
App	Main application state and orchestration
Header	Application branding and navigation
Toolbar	Tool selection and graph settings
GraphCanvas	Interactive graph rendering with pan/zoom
AlgorithmPanel	Algorithm selection and playback controls
MatrixDisplay	Adjacency and distance matrix visualization
StepLog	Step-by-step algorithm execution log
About	Team and project information modal

3.4 Data Structures

3.4.1 Graph Representation

```
1 interface Graph {
2     nodes: Node[];
3     edges: Edge[];
4     isDirected: boolean;
5     isWeighted: boolean;
6 }
```

```
7
8 interface Node {
9   id: string;
10  label: string;
11  x: number;
12  y: number;
13 }
14
15 interface Edge {
16   id: string;
17   source: string;
18   target: string;
19   weight: number;
20 }
```

3.4.2 Algorithm Step

```
1 interface AlgorithmStep {
2   type: 'visit' | 'explore' | 'update' | 'complete';
3   nodeId?: string;
4   edgeId?: string;
5   message: string;
6   visited?: string[];
7   currentPath?: string[];
8   distances?: Map<string, number>;
9 }
```

3.5 State Management

The application uses React's built-in state management with the following key state variables:

- **Graph State:** Current graph structure
- **Selection State:** Selected nodes and edges
- **Tool State:** Current active tool
- **Visualization State:** Algorithm execution progress

- **UI State:** Panel visibility and settings

Chapter 4

Implementation

4.1 Project Structure

```
GraphAlgorithmVisualize/
  public/
    favicon.svg
  src/
    components/
      About.tsx
      AlgorithmPanel.tsx
      GraphCanvas.tsx
      Header.tsx
      MatrixDisplay.tsx
      StepLog.tsx
      Toolbar.tsx
    types/
      index.ts
    utils/
      algorithmInfo.ts
      algorithms.ts
      graphUtils.ts
  App.tsx
```

```

index.css
main.tsx
docs/
    report.tex
.github/
    workflows/
        deploy.yml
index.html
package.json
tailwind.config.js
tsconfig.json
vite.config.ts

```

4.2 Key Features Implementation

4.2.1 Interactive Canvas

The GraphCanvas component implements:

- **Node Operations:** Add, move, delete, select
- **Edge Operations:** Create connections, set weights
- **Pan and Zoom:** Navigate large graphs
- **Visual Feedback:** Color-coded states and animations

Coordinate Transformation

```

1 const screenToCanvas = (screenX: number, screenY: number) => {
2     return {
3         x: (screenX - pan.x) / zoom,
4         y: (screenY - pan.y) / zoom,
5     };
6 }

```

4.2.2 Algorithm Visualization

Each algorithm returns a sequence of steps that are animated:

```

1 interface AlgorithmResult {
2     steps: AlgorithmStep[];
3     path?: string[];
4     distances?: Map<string, number>;
5 }
```

The visualization system:

1. Executes the algorithm and records steps
2. Animates through steps at configurable speed
3. Updates node/edge colors based on state
4. Displays step-by-step explanations

4.2.3 Visual State Colors

Table 4.1: Node Color States

State	Color	Hex Code
Default	Slate	#475569
Selected	Purple	#d946ef
Visited	Blue	#0ea5e9
Current	Orange	#f97316
Path	Green	#22c55e

4.3 Algorithm Implementations

```

1 function bfs(graph: Graph, startId: string): AlgorithmResult {
2     const steps: AlgorithmStep[] = [];
3     const visited = new Set<string>();
4     const queue: string[] = [startId];
5     const parent = new Map<string, string | null>();
```

```

7   visited.add(startId);
8   parent.set(startId, null);
9
10  while (queue.length > 0) {
11    const current = queue.shift()!;
12    steps.push({
13      type: 'visit',
14      nodeId: current,
15      message: `Visiting node ${getLabel(current)}`,
16      visited: Array.from(visited)
17    });
18
19    for (const neighbor of getNeighbors(graph, current)) {
20      if (!visited.has(neighbor.id)) {
21        visited.add(neighbor.id);
22        parent.set(neighbor.id, current);
23        queue.push(neighbor.id);
24      }
25    }
26  }
27
28  return { steps };
29}

```

4.3.2 Dijkstra Implementation

```

1 function dijkstra(graph: Graph, startId: string): AlgorithmResult {
2   const steps: AlgorithmStep[] = [];
3   const distances = new Map<string, number>();
4   const previous = new Map<string, string | null>();
5   const unvisited = new Set<string>();
6
7   // Initialize
8   for (const node of graph.nodes) {
9     distances.set(node.id, Infinity);
10    previous.set(node.id, null);
11    unvisited.add(node.id);
12  }
13  distances.set(startId, 0);

```

```

14
15   while (unvisited.size > 0) {
16     // Find minimum distance node
17     let current = null;
18     let minDist = Infinity;
19     for (const id of unvisited) {
20       if (distances.get(id)! < minDist) {
21         minDist = distances.get(id)!;
22         current = id;
23       }
24     }
25
26     if (current === null) break;
27     unvisited.delete(current);
28
29     // Update neighbors
30     for (const edge of getEdgesFrom(graph, current)) {
31       const newDist = distances.get(current)! + edge.weight;
32       if (newDist < distances.get(edge.target)!) {
33         distances.set(edge.target, newDist);
34         previous.set(edge.target, current);
35       }
36     }
37   }
38
39   return { steps, distances };
40 }
```

4.3.3 DFS Implementation

```

1 function dfs(graph: Graph, startId: string,
2             detectCycles: boolean = true): AlgorithmResult {
3   const steps: AlgorithmStep[] = [];
4   const visited = new Set<string>();
5   const recursionStack = new Set<string>();
6   const parents = new Map<string, string | null>();
7   let hasCycle = false;
8
9   function dfsRecursive(nodeId: string,
```

```
10          parentId: string | null): boolean {
11
12    visited.add(nodeId);
13
14    recursionStack.add(nodeId);
15
16    parents.set(nodeId, parentId);
17
18    const node = getNodeById(graph, nodeId);
19
20
21    steps.push({
22      type: 'visit',
23      nodeId,
24      message: `Visiting node ${node?.label}`,
25      visited: new Set(visited),
26      stack: Array.from(recursionStack),
27    });
28
29
30    const neighbors = getNeighbors(graph, nodeId);
31
32    for (const { node: neighbor, edge } of neighbors) {
33
34      steps.push({
35        type: 'explore',
36        nodeId: neighbor.id,
37        edgeId: edge.id,
38        message: `Exploring edge to ${neighbor.label}`,
39        visited: new Set(visited),
40        stack: Array.from(recursionStack),
41      });
42
43
44      if (!visited.has(neighbor.id)) {
45
46        if (dfsRecursive(neighbor.id, nodeId)) {
47
48          return true;
49        }
50
51      } else if (detectCycles && recursionStack.has(neighbor.id)) {
52
53        // Cycle detected
54
55        hasCycle = true;
56
57        steps.push({
58          type: 'cycle-detected',
59          nodeId: neighbor.id,
60          edgeId: edge.id,
61          message: `Cycle detected at node ${neighbor.label}!`,
62          visited: new Set(visited),
63        });
64
65      }
66
67    }
68
69  }
70
71
72  if (!hasCycle) {
73
74    return true;
75  }
76
77  return false;
78}
```

```

49         stack: Array.from(recursionStack) ,
50     }) ;
51     return true;
52 }
53 }
54
55 recursionStack.delete(nodeId);
56 steps.push({
57     type: 'backtrack',
58     nodeId,
59     message: 'Backtracking from node ${node?.label}',
60     visited: new Set(visited),
61     stack: Array.from(recursionStack),
62 }) ;
63
64 return false;
65 }
66
67 dfsRecursive(startId, null);
68 return { steps, hasCycle };
69 }
```

4.3.4 Bellman-Ford Implementation

```

1 function bellmanFord(graph: Graph, startId: string,
2                         endId?: string): AlgorithmResult {
3     const steps: AlgorithmStep[] = [];
4     const distances = new Map<string, number>();
5     const parents = new Map<string, string | null>();
6     const n = graph.nodes.length;
7
8     // Initialize distances
9     for (const node of graph.nodes) {
10         distances.set(node.id, node.id === startId ? 0 : Infinity);
11     }
12     parents.set(startId, null);
13
14     steps.push({
15         type: 'visit',
```

```

16     nodeId: startId,
17     message: `Starting Bellman-Ford from ${getNodeById(graph, startId)
18       ?.label}`,
19     distances: new Map(distances),
20   });
21
22   // Relax edges n-1 times
23   for (let i = 0; i < n - 1; i++) {
24     let updated = false;
25
26     steps.push({
27       type: 'visit',
28       message: `Iteration ${i + 1} of ${n - 1}`,
29       distances: new Map(distances),
30     });
31
32     for (const edge of graph.edges) {
33       const edgesToCheck = graph.isDirected
34         ? [{from: edge.source, to: edge.target}]
35         : [{from: edge.source, to: edge.target},
36              {from: edge.target, to: edge.source}];
37
38       for (const {from, to} of edgesToCheck) {
39         const fromDist = distances.get(from)!;
40         const toDist = distances.get(to)!;
41
42         if (fromDist !== Infinity && fromDist + edge.weight < toDist)
43         {
44           distances.set(to, fromDist + edge.weight);
45           parents.set(to, from);
46           updated = true;
47
48           steps.push({
49             type: 'relax',
50             nodeId: to,
51             edgeId: edge.id,
52             message: `Relaxed edge: distance to ${getNodeById(graph,
53               to)?.label} = ${fromDist + edge.weight}`,
54             distances: new Map(distances),
55           });
56         }
57       }
58     }
59   }
60 }
```

```
52         });
53     }
54   }
55 }
56
57 if (!updated) {
58   steps.push({
59     type: 'visit',
60     message: `No updates in iteration ${i + 1}, terminating early
61   `,
62     distances: new Map(distances),
63   });
64   break;
65 }
66
67 // Check for negative cycles
68 let hasNegativeCycle = false;
69 for (const edge of graph.edges) {
70   const edgesToCheck = graph.isDirected
71     ? [{ from: edge.source, to: edge.target }]
72     : [{ from: edge.source, to: edge.target },
73       { from: edge.target, to: edge.source }];
74
75   for (const { from, to } of edgesToCheck) {
76     const fromDist = distances.get(from)!;
77     const toDist = distances.get(to)!;
78
79     if (fromDist !== Infinity && fromDist + edge.weight < toDist) {
80       hasNegativeCycle = true;
81       steps.push({
82         type: 'cycle-detected',
83         edgeId: edge.id,
84         message: 'Negative cycle detected!',
85         distances: new Map(distances),
86       });
87     }
88   }
89 }
```

```

90
91     if (hasNegativeCycle) {
92         steps.push({
93             type: 'complete',
94             message: 'Algorithm complete - Negative cycle exists!',
95             distances: new Map(distances),
96         });
97         return { steps, hasNegativeCycle: true, distances };
98     }
99
100    steps.push({
101        type: 'complete',
102        message: 'Bellman-Ford complete',
103        distances: new Map(distances),
104    });
105
106    return { steps, distances, hasNegativeCycle: false };
107 }
```

4.3.5 A* Implementation

```

1 function aStar(graph: Graph, startId: string,
2                 endId: string): AlgorithmResult {
3     const steps: AlgorithmStep[] = [];
4     const startNode = getNodeById(graph, startId)!;
5     const endNode = getNodeById(graph, endId)!;
6
7     // Heuristic: Euclidean distance
8     const heuristic = (nodeId: string): number => {
9         const node = getNodeById(graph, nodeId)!;
10        const dx = node.x - endNode.x;
11        const dy = node.y - endNode.y;
12        return Math.sqrt(dx * dx + dy * dy) / 50; // Scale factor
13    };
14
15    const gScore = new Map<string, number>();
16    const fScore = new Map<string, number>();
17    const parents = new Map<string, string | null>();
18    const openSet: string[] = [startId];
```

```

19   const closedSet = new Set<string>();
20
21   for (const node of graph.nodes) {
22     gScore.set(node.id, node.id === startId ? 0 : Infinity);
23     fScore.set(node.id, node.id === startId ? heuristic(startId) :
24     Infinity);
25   }
26
27   parents.set(startId, null);
28
29   steps.push({
30     type: 'visit',
31     nodeId: startId,
32     message: `Starting A* from ${startNode.label} to ${endNode.label}
33     }`,
34     distances: new Map(gScore),
35   });
36
37   while (openSet.length > 0) {
38     // Get node with lowest fScore
39     openSet.sort((a, b) => (fScore.get(a) || Infinity) -
40                   (fScore.get(b) || Infinity));
41     const currentId = openSet.shift()!;
42     const currentNode = getNodeById(graph, currentId)!;
43
44     closedSet.add(currentId);
45
46     steps.push({
47       type: 'visit',
48       nodeId: currentId,
49       message: `Processing ${currentNode.label} (g=${gScore.get(
50         currentId)?.toFixed(1)}, f=${fScore.get(currentId)?.toFixed(1)})`,
51       distances: new Map(gScore),
52       visited: new Set(closedSet),
53     });
54
55     if (currentId === endId) {
56       const path = reconstructPath(parents, endId);
57       steps.push({
58         type: 'complete',
59       });
60     }
61   }
62
63   return steps;
64 }

```

```

55     message: `Path found! Distance: ${gScore.get(endId)?.toFixed
56     (1)}`,
57     currentPath: path,
58     distances: new Map(gScore),
59   );
60   return { steps, shortestPath: path, distances: gScore };
61 }
62
63 const neighbors = getNeighbors(graph, currentId);
64 for (const { node, edge } of neighbors) {
65   if (closedSet.has(node.id)) continue;
66
67   const weight = edge.weight || 1;
68   const tentativeG = (gScore.get(currentId) || 0) + weight;
69
70   steps.push({
71     type: 'explore',
72     nodeId: node.id,
73     edgeId: edge.id,
74     message: `Exploring ${node.label}: g=${tentativeG.toFixed(1)}, h=${heuristic(node.id).toFixed(1)}`,
75     distances: new Map(gScore),
76   });
77
78   if (tentativeG < (gScore.get(node.id) || Infinity)) {
79     parents.set(node.id, currentId);
80     gScore.set(node.id, tentativeG);
81     fScore.set(node.id, tentativeG + heuristic(node.id));
82
83     if (!openSet.includes(node.id)) {
84       openSet.push(node.id);
85     }
86
87     steps.push({
88       type: 'relax',
89       nodeId: node.id,
90       edgeId: edge.id,
91       message: `Updated ${node.label}: g=${tentativeG.toFixed(1)}, f=${fScore.get(node.id)?.toFixed(1)}`,
92     });
93   }
94 }
95
96 return { steps, shortestPath: path, distances: gScore };

```

```
91         distances: new Map(gScore),
92         });
93     }
94   }
95 }
96
97 steps.push({
98   type: 'complete',
99   message: 'No path found!',
100  distances: new Map(gScore),
101 });
102
103 return { steps, distances: gScore };
104 }
```

4.4 Deployment Configuration

4.4.1 GitHub Actions Workflow

```
1 name: Deploy to GitHub Pages
2
3 on:
4   push:
5     branches: [ main ]
6
7 jobs:
8   build:
9     runs-on: ubuntu-latest
10  steps:
11    - uses: actions/checkout@v4
12    - uses: actions/setup-node@v4
13      with:
14        node-version: '20'
15    - run: npm ci
16    - run: npm run build
17    - uses: actions/upload-pages-artifact@v3
18      with:
19        path: ./dist
```

```
20  
21   deploy:  
22     needs: build  
23     runs-on: ubuntu-latest  
24   steps:  
25     - uses: actions/deploy-pages@v4
```

Chapter 5

User Guide

5.1 Getting Started

5.1.1 Accessing the Application

The application is available at:

<https://hambozo17.github.io/GraphAlgorithmVisualize/>

5.1.2 Interface Overview

The interface consists of:

1. **Header:** Application title and About button
2. **Toolbar:** Tools and graph settings
3. **Canvas:** Main graph editing area
4. **Algorithm Panel:** Algorithm selection and controls
5. **Bottom Panel:** Matrix display and step log

5.1.3 Application Interface

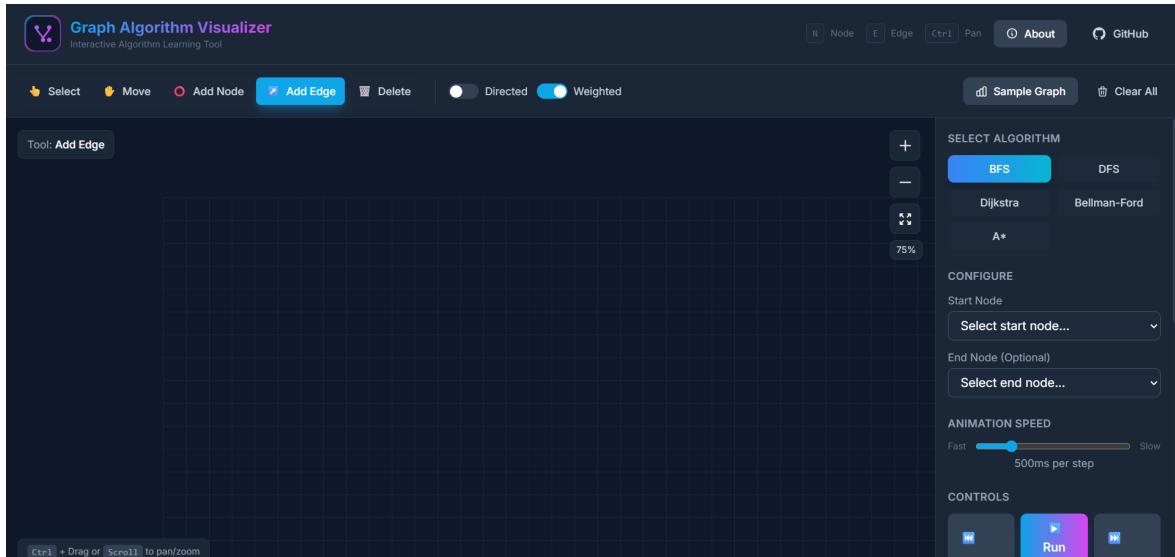


Figure 5.1: Graph Algorithm Visualizer main interface showing the interactive canvas, toolbar, and algorithm panel

5.2 Creating a Graph

5.2.1 Adding Nodes

1. Select the "Add Node" tool (or press N)
2. Click anywhere on the canvas to place a node
3. Nodes are automatically labeled (A, B, C, ...)

5.2.2 Adding Edges

1. Select the "Add Edge" tool (or press E)
2. Click on the source node
3. Click on the target node
4. For weighted graphs, enter the weight when prompted

5.2.3 Graph Settings

- **Directed:** Toggle arrow direction on edges

- **Weighted:** Enable edge weights

5.3 Running Algorithms

5.3.1 Setup

1. Select an algorithm from the dropdown
2. Choose a start node (click while in Select mode)
3. For A*, also select an end node

5.3.2 Playback Controls

- **Play:** Start/resume animation
- **Pause:** Pause animation
- **Step Forward:** Advance one step
- **Step Backward:** Go back one step
- **Stop:** Reset visualization
- **Speed Slider:** Adjust animation speed

5.4 Keyboard Shortcuts

Table 5.1: Keyboard Shortcuts

Key	Action
V	Select tool
M	Move tool
N	Add Node tool
E	Add Edge tool
D / Delete	Delete tool
Space	Play/Pause animation
Escape	Stop animation
Ctrl + Drag	Pan canvas
Scroll	Zoom in/out

5.5 Pan and Zoom

- **Zoom:** Use scroll wheel or +/- buttons
- **Pan:** Hold Ctrl and drag, or use middle mouse button
- **Reset:** Click the reset view button

Chapter 6

Testing and Validation

6.1 Test Cases

6.1.1 Graph Operations

Table 6.1: Graph Operation Tests

Test Case	Expected Result	Status
Add node	Node appears at click position	Pass
Add edge	Edge connects two nodes	Pass
Delete node	Node and connected edges removed	Pass
Move node	Node position updates	Pass
Toggle directed	Arrows appear/disappear	Pass
Toggle weighted	Weight labels shown/hidden	Pass

6.1.2 Algorithm Tests

Table 6.2: Algorithm Test Results

Algorithm	Test Graph	Expected	Status
BFS	5-node connected	All nodes visited	Pass
DFS	5-node connected	All nodes visited	Pass
Dijkstra	Weighted graph	Shortest path found	Pass
Bellman-Ford	Negative weights	Correct distances	Pass
A*	Grid-like graph	Optimal path found	Pass

6.2 Browser Compatibility

Table 6.3: Browser Compatibility

Browser	Version	Status
Google Chrome	120+	Fully Compatible
Mozilla Firefox	120+	Fully Compatible
Microsoft Edge	120+	Fully Compatible
Safari	17+	Fully Compatible

Chapter 7

Conclusion

7.1 Summary

The Graph Algorithm Visualizer successfully achieves its objectives of providing an interactive, educational tool for understanding graph algorithms. The application enables users to:

- Create and manipulate graphs intuitively
- Visualize algorithm execution step-by-step
- Understand the differences between various algorithms
- Learn through hands-on experimentation

7.2 Key Achievements

1. **Comprehensive Algorithm Coverage:** Five fundamental algorithms implemented
2. **Interactive Visualization:** Real-time, animated algorithm execution
3. **Modern User Interface:** Clean, responsive design with dark theme
4. **Educational Value:** Step-by-step explanations and complexity information
5. **Accessibility:** Web-based, no installation required

7.3 Future Enhancements

Potential improvements for future versions:

- Additional algorithms (Prim's, Kruskal's, Floyd-Warshall)
- Graph import/export functionality
- Mobile-responsive design
- Collaborative editing features
- Algorithm comparison mode
- Custom graph templates

7.4 Lessons Learned

This project provided valuable experience in:

- Modern web development with React and TypeScript
- Algorithm implementation and visualization
- User interface design principles
- Project collaboration and version control
- Continuous deployment with GitHub Actions

Bibliography

- [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.
- [2] Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.
- [3] React Documentation. <https://react.dev/>
- [4] TypeScript Documentation. <https://www.typescriptlang.org/docs/>
- [5] Vite Documentation. <https://vitejs.dev/>
- [6] Tailwind CSS Documentation. <https://tailwindcss.com/docs>
- [7] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 269-271.
- [8] Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1), 87-90.
- [9] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.

Appendix A

Source Code Repository

The complete source code is available at:

```
https://github.com/Hambozo17/GraphAlgorithmVisualize
```

A.1 Installation Instructions

1. Clone the repository:

```
git clone https://github.com/Hambozo17/GraphAlgorithmVisualize.git
```

2. Install dependencies:

```
cd GraphAlgorithmVisualize  
npm install
```

3. Start development server:

```
npm run dev
```

4. Build for production:

```
npm run build
```

Appendix B

Team Contributions

Table B.1: Team Member Contributions

Team Member	Contributions
Mohamed Mohsen (221001411)	Algorithm implementations, Canvas rendering, State management
Farah Khaled (221001643)	Algorithm implementations, UI/UX design, Component development, Testing
Omar Ahmed (221001335)	Algorithm implementations, Documentation, Deployment, Code review