Hanoi University of Science and Technology

School of Information and Communication Technology

Kruskal MST Visualizer

An Interactive Visualization Tool for Kruskal's Algorithm

Project I Report

Student: Hoang Van Nhan
Student ID: 20235542
Instructor: Ms. Nguyen Khanh Phuong

Hanoi, January 2026

Contents

1 Introduction

1.1 Background and Motivation

Graph algorithms are fundamental concepts in computer science education. Among these, Kruskal's algorithm for finding the Minimum Spanning Tree (MST) is one of the most important algorithms taught in data structures and algorithms courses. However, understanding how this algorithm works step-by-step can be challenging for students, especially when dealing with the underlying data structures like Disjoint Set Union (DSU) or cycle detection using Depth-First Search (DFS).

This project was developed to address this educational challenge by creating an interactive web-based visualization tool that helps students and developers understand how Kruskal's algorithm works through step-by-step visualization.

1.2 Project Objectives

The main objectives of this project are:

1. Develop an interactive web application that visualizes Kruskal's MST algorithm
2. Support two different implementation approaches:
   - Kruskal + DSU (Disjoint Set Union with Union by Rank and Path Compression)
   - Kruskal + DFS (using Depth-First Search for cycle detection)
3. Provide step-by-step execution with detailed explanations
4. Create a responsive design that works on both desktop and mobile devices
5. Apply modern web development technologies and best practices

1.3 Deployment

The application is deployed and publicly accessible at:

`https://kruskal-mst-visualizer.vercel.app/`

2 Theoretical Background

2.1 Minimum Spanning Tree Problem

Given a connected, undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, a Minimum Spanning Tree (MST) is a subset of edges $T \subseteq E$ such that:

1. T connects all vertices (forms a spanning tree)
2. The sum $\sum_{e \in T} w(e)$ is minimized

2.2 Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm that finds the MST by:

1. Sorting all edges in non-decreasing order of weight
2. Iterating through each edge and adding it to the MST if it doesn't create a cycle
3. Stopping when the MST has $|V| - 1$ edges

Algorithm 1 Kruskal's Algorithm

function KRUSKAL($G = (V, E)$)
    Sort E by weight in ascending order
    $T \leftarrow \emptyset$ ▷ MST edges
    for each edge (u, v, w) in sorted order **do**
        if adding (u, v) to T doesn't create a cycle **then**
            $T \leftarrow T \cup \{(u, v)\}$
        end if
    end for
    return T
end function

The key challenge is efficiently detecting whether adding an edge would create a cycle. This project implements two approaches:

2.3 Disjoint Set Union (DSU)

DSU (also known as Union-Find) is a data structure that efficiently handles:

- `find(x)`: Find the representative (root) of the set containing x
- `union(x, y)`: Merge the sets containing x and y

With **Union by Rank** and **Path Compression** optimizations, both operations achieve nearly $O(\alpha(n))$ amortized time complexity, where α is the inverse Ackermann function (practically constant).

2.3.1 Python Implementation with Path Compression

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n + 1))
        self.rank = [0] * (n + 1)

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path
    compression
        return self.parent[u]

    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
```

```
        if pu == pv:
            return False
        if self.rank[pu] < self.rank[pv]:
            pu, pv = pv, pu
        self.parent[pv] = pu
        if self.rank[pu] == self.rank[pv]:
            self.rank[pu] += 1
        return True

def kruskal(n, edges):
    edges.sort(key=lambda e: e[2])
    dsu = DSU(n)
    mst_weight = 0
    mst_edges = []
    for u, v, w in edges:
        if dsu.union(u, v):
            mst_weight += w
            mst_edges.append((u, v, w))
    return mst_weight, mst_edges
```

Listing 1: DSU with Path Compression

2.3.2 Python Implementation without Path Compression

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n + 1))
        self.rank = [0] * (n + 1)

    def find(self, u):
        if self.parent[u] != u:
            return self.find(self.parent[u])  # No path compression
        return self.parent[u]

    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
        if pu == pv:
            return False
        if self.rank[pu] < self.rank[pv]:
            pu, pv = pv, pu
        self.parent[pv] = pu
        if self.rank[pu] == self.rank[pv]:
            self.rank[pu] += 1
        return True
```

Listing 2: DSU without Path Compression

2.4 DFS-based Cycle Detection

An alternative approach uses DFS to check if there's already a path between two vertices in the current MST before adding an edge. This method is simpler but less efficient with $O(V + E)$ per edge check.

```
def has_path(u, v, adj, visited):
    if u == v:
        return True
```

```python

    visited.add(u)
    for x in adj[u]:
        if x not in visited and has_path(x, v, adj, visited):
            return True
    return False

def kruskal_dfs(n, edges):
    edges.sort(key=lambda e: e[2])
    adj = {i: [] for i in range(1, n + 1)}
    mst_weight = 0
    mst_edges = []
    for u, v, w in edges:
        visited = set()
        if not has_path(u, v, adj, visited):
            adj[u].append(v)
            adj[v].append(u)
            mst_weight += w
            mst_edges.append((u, v, w))
    return mst_weight, mst_edges
```

Listing 3: Kruskal with DFS-based cycle detection

3 System Design

3.1 Technology Stack

The project utilizes modern web development technologies:

Table 1: Technology Stack

Category	Technology	Purpose
Frontend Framework	React 19	UI component library
Programming Language	TypeScript	Type-safe JavaScript
Build Tool	Vite (Rolldown)	Fast development & bundling
Graph Visualization	Sigma.js + Graphology	Interactive graph rendering
Code Highlighting	Prism React Renderer	Syntax highlighting
Drag & Drop	@dnd-kit	Panel reordering
Deployment	Vercel	Hosting & CI/CD
Analytics	Vercel Speed Insights	Performance monitoring

3.2 Project Structure

```
kruskal-mst-visualizer/
|-- src/
|   |-- components/          # React UI components
|   |   |-- CodeViewer.tsx       # Python code with highlighting
|   |   |-- CollapsiblePanel.tsx # Collapsible wrapper
|   |   |-- ControlPanel.tsx     # Playback controls
|   |   |-- DFSPanel.tsx         # DFS state visualization
|   |   |-- DraggablePanel.tsx   # Drag & drop wrapper
```

```
|   |   |-- DSUPanel.tsx          # DSU state table
|   |   |-- EdgeListPanel.tsx     # Sorted edge list
|   |   |-- ExplanationPanel.tsx  # Step explanations
|   |   |-- GraphInput.tsx        # Graph input form
|   |   |-- GraphRenderer.tsx     # Sigma.js visualization
|   |   |-- HelpModal.tsx         # Color legend modal
|   |-- engine/                 # Algorithm implementations
|   |   |-- kruskalDsu.ts         # Kruskal + DSU step generator
|   |   |-- kruskalDfs.ts         # Kruskal + DFS step generator
|   |   |-- parser.ts             # Graph input parser
|   |   |-- types.ts              # TypeScript types
|   |-- App.tsx                 # Main application
|   |-- App.css                 # Styles
|   |-- main.tsx                # Entry point
|-- scripts/
|   |-- test-all.ts             # Automated tests
|-- test-cases/                 # Test input/output files
|-- package.json
```

Listing 4: Project directory structure

3.3 Architecture Overview

The application follows a component-based architecture with clear separation of concerns:

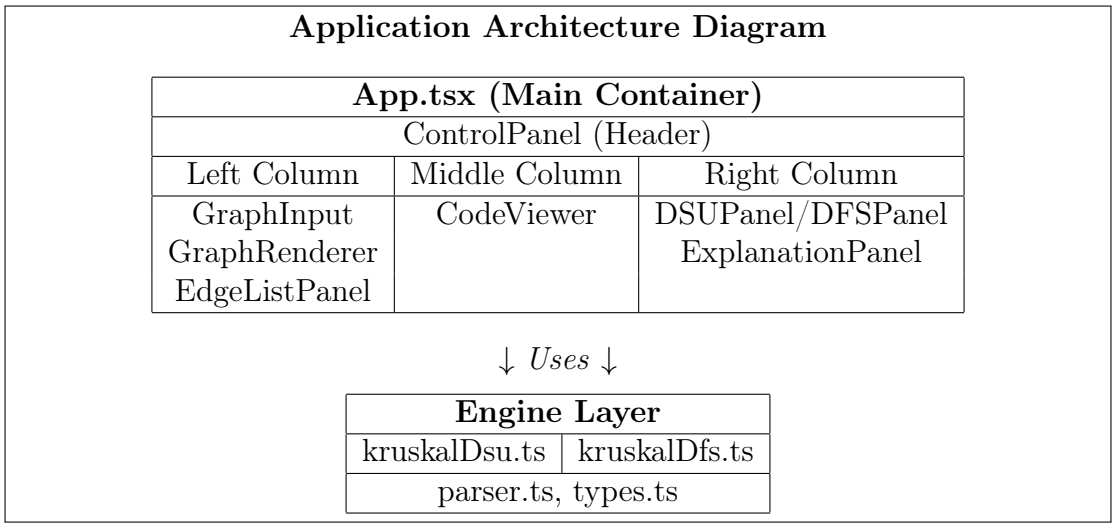

Figure 1: High-level application architecture

3.4 Core Data Types

The application uses well-defined TypeScript types for type safety:

```
export type EdgeStatus = 'normal' | 'chosen' | 'rejected' | 'current';

export type NodeStatus =
    | "normal"
    | "findStart"
    | "findWalk"
```

```
    | "findRoot"
    | "dfsCurrent"
    | "dfsCandidate"
    | "dfsVisited";

export interface Edge {
    id: string;   // e.g., "e0", "e1", ...
    u: number;    // start vertex
    v: number;    // end vertex
    w: number;    // weight
}

export interface Step {
    stepId: number;
    stepLabel?: string;                    // "3.1.2"
    currentEdge: Edge | null;
    edgeStatus: Record<string, EdgeStatus>;
    sortedEdgeIds?: string[];
    mstEdges: string[];
    highlightedLines: number[];
    explanation: string;
    mstWeight?: number;

    // DSU state
    dsuParent?: number[];
    dsuRank?: number[];
    focusNodes?: number[];
    nodeStatus?: Record<string, NodeStatus>;

    // DFS state
    dfsSource?: number;
    dfsTarget?: number;
    dfsCurrent?: number;
    dfsVisited?: number[];
    dfsStack?: number[];
    dfsEdgeOverlay?: Record<string, "active" | "dead" | "candidate">;
}
```

Listing 5: Core TypeScript types (types.ts)

4 Implementation Details

4.1 Step Generation Engine

The core of the visualization is the step generation engine, which simulates the algorithm execution and generates a sequence of `Step` objects that capture the algorithm state at each point.

4.1.1 DSU Step Generator

The DSU step generator (`kruskalDsu.ts`) creates detailed steps for:

- Initialization step (sorting edges, initializing DSU)
- For each edge:

  - Edge consideration step
  - `find(u)` operation steps (with optional detailed hop-by-hop trace)
  - `find(v)` operation steps
  - Path compression steps (when enabled)
  - Union or rejection decision step
- Final result step

Key implementation features:

```
export interface DsuSimOptions {
    detailed: boolean;       // coarse vs detailed mode
    compression: boolean;    // path compression ON/OFF
    maxFindHops: number;     // limit steps per find() to avoid
   explosion
}

export function buildKruskalDsuSteps(
    graph: ParsedGraph,
    opts: Partial<DsuSimOptions> = {}
): Step[] {
    const options: DsuSimOptions = {
        detailed: opts.detailed ?? false,
        compression: opts.compression ?? true,
        maxFindHops: Math.max(1, opts.maxFindHops ?? 6),
    };

    // ... step generation logic

    // Step labeling system: major.minor.sub
    // e.g., "3.1.2" = 3rd edge, 1st find, 2nd hop

    return steps;
}
```

Listing 6: DSU step generator structure

4.1.2 DFS Step Generator

The DFS step generator (`kruskalDfs.ts`) creates steps for:

- Initialization step
- For each edge:
  - Edge consideration step
  - DFS traversal steps (enter node, explore neighbors, backtrack)
  - Result step (path found or not found)
- Final result step

The DFS overlay system provides visual feedback on the graph:

```
dfsEdgeOverlay?: Record<string, "active" | "dead" | "candidate">;

// "active"    - Current path being explored (cyan)
// "dead"      - Backtracked, failed path (gray)
// "candidate" - Potential neighbors to explore (orange)
```

Listing 7: DFS edge overlay types

4.2 Graph Visualization with Sigma.js

The `GraphRenderer` component uses Sigma.js and Graphology for interactive graph visualization:

```
// Circular layout for nodes
const radius = Math.max(100, nodeCount * 8);
nodes.forEach((nodeId, index) => {
    const angle = (2 * Math.PI * index) / nodeCount;
    const x = radius * Math.cos(angle);
    const y = radius * Math.sin(angle);

    graph.addNode(String(nodeId), {
        x, y,
        size: 30 / Math.sqrt(nodeCount),
        label: String(nodeId),
        color: "#0f172a"
    });
});

// Edge coloring based on status
const statusColors = {
    normal: "#1e3260",
    current: "#facc15",  // Yellow
    chosen: "#22c55e",   // Green
    rejected: "#6b7280"  // Gray
};
```

Listing 8: Graph initialization with circular layout

4.3 Drag and Drop Panel System

The application uses @dnd-kit for draggable panels, with different behavior for desktop and mobile:

```
// Desktop (>1024px): Drag within columns
const [leftPanelOrder, setLeftPanelOrder] = useState([
    "graph-input", "graph-viz", "edge-list"
]);
const [middlePanelOrder, setMiddlePanelOrder] = useState(["code"]);
const [rightPanelOrder, setRightPanelOrder] = useState([
    "state", "explanation"
]);

// Mobile (<=1024px): Single column, free dragging
const [singleColumnOrder, setSingleColumnOrder] = useState([
    "graph-input", "graph-viz", "edge-list",
    "code", "state", "explanation"
```

```
]);
```

Listing 9: Drag and drop configuration

4.4 Responsive Design

The application adapts to different screen sizes:

- **Desktop (>1024px)**: Three-column layout with panel dragging within columns
- **Tablet/Mobile (≤1024px)**: Single-column layout with free panel reordering
- **Mobile (≤640px)**: Collapsible panels default to collapsed state except essential ones

```
const [isMobile, setIsMobile] = useState(() => window.innerWidth <= 640)
    ;
const [isSmallScreen, setIsSmallScreen] = useState(() =>
    window.innerWidth <= 1024
);

useEffect(() => {
    const handleResize = () => {
        setIsMobile(window.innerWidth <= 640);
        setIsSmallScreen(window.innerWidth <= 1024);
    };
    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
}, []);
```

Listing 10: Responsive breakpoint detection

5 User Interface

5.1 Main Interface

The main interface consists of several panels organized in a flexible layout:

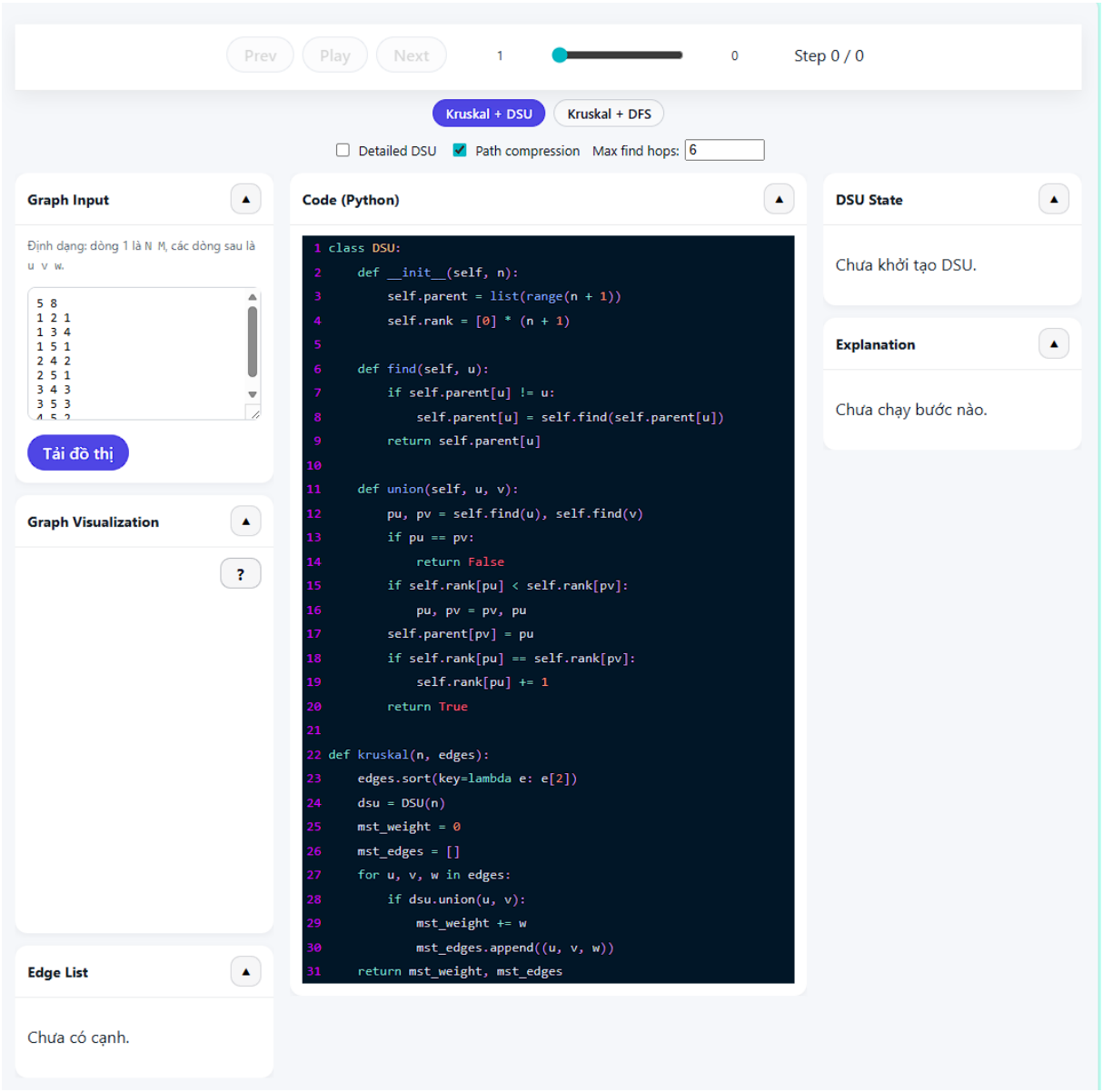

Figure 2: Main interface showing graph visualization, code panel, and explanation

5.2 Visual Feedback System

The application uses color coding to convey algorithm state:

5.2.1 Edge States

- **Yellow**: Current edge being considered
- **Green**: Edge chosen for MST
- **Gray**: Edge rejected (would create cycle)

5.2.2 DSU Node States

- **Cyan**: Node calling `find(u)` / Root found

- **Yellow**: Node being traversed during find operation
- **Purple**: Root node of the set

5.2.3 DFS Overlay

- **Orange**: Candidate neighbors to explore
- **Cyan**: Active DFS path
- **Gray**: Dead branch (backtracked)

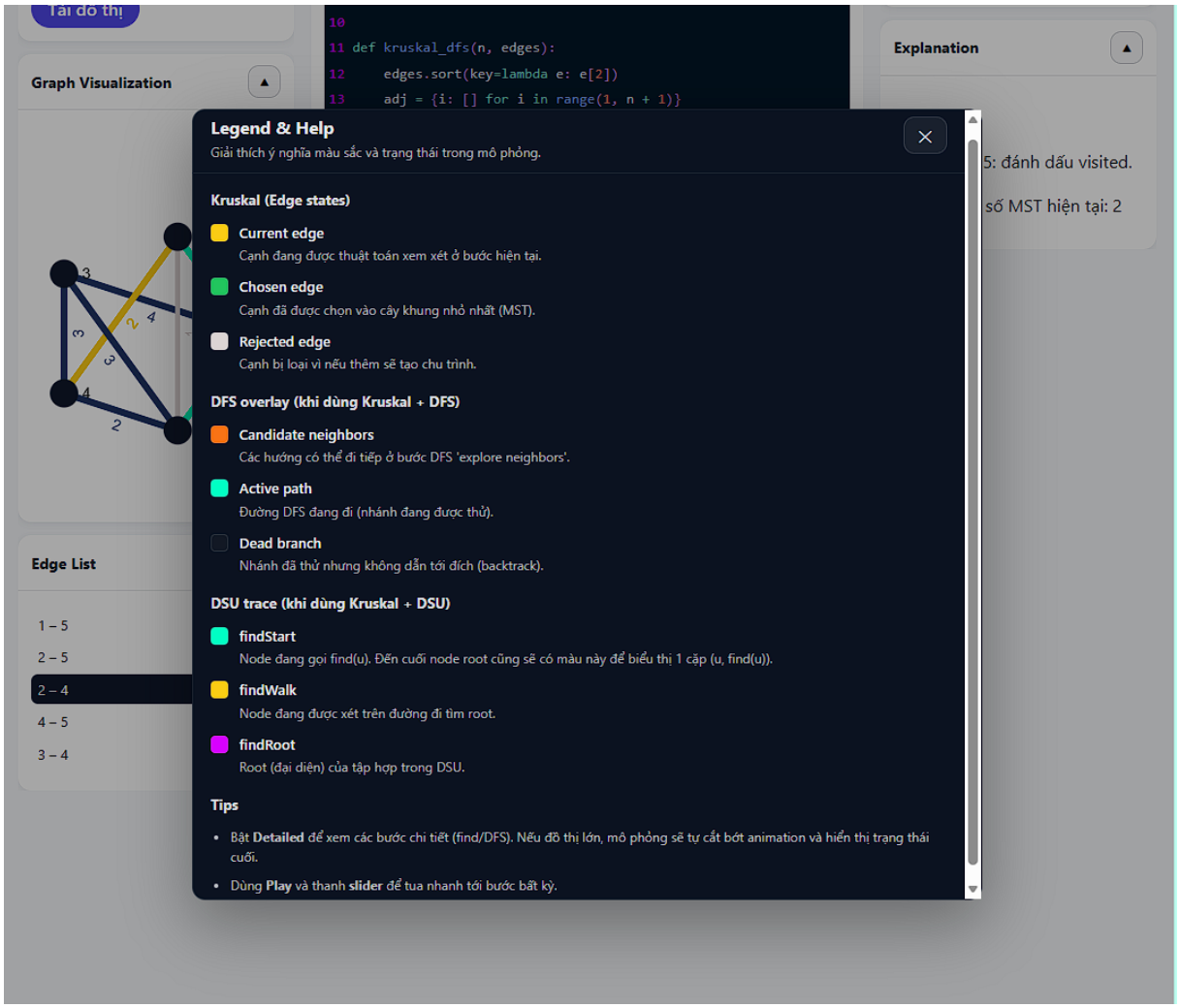

Figure 3: Help modal explaining all color codes

5.3 DSU Mode Visualization

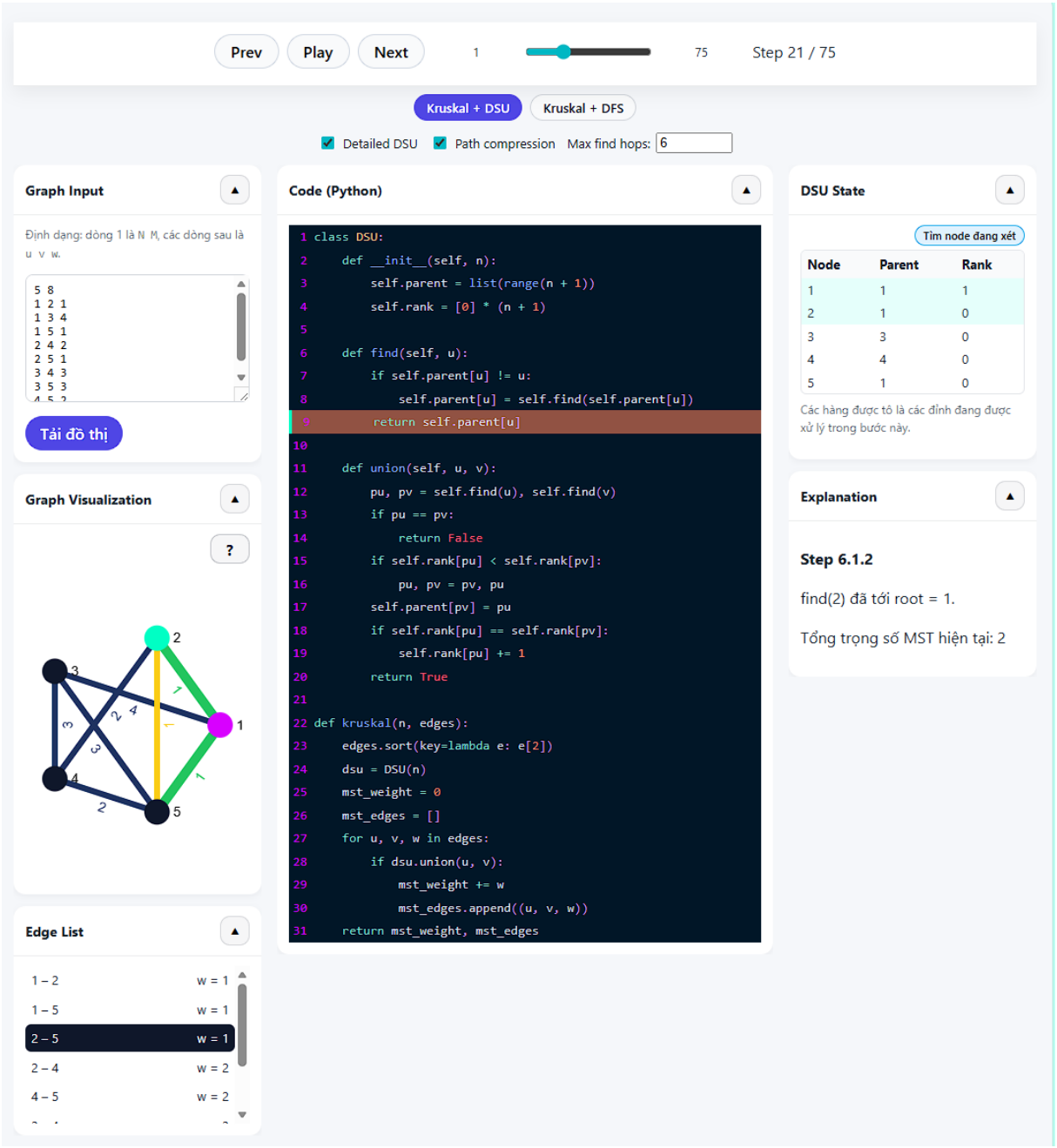

Figure 4: DSU mode showing parent/rank values and path compression

5.4 DFS Mode Visualization

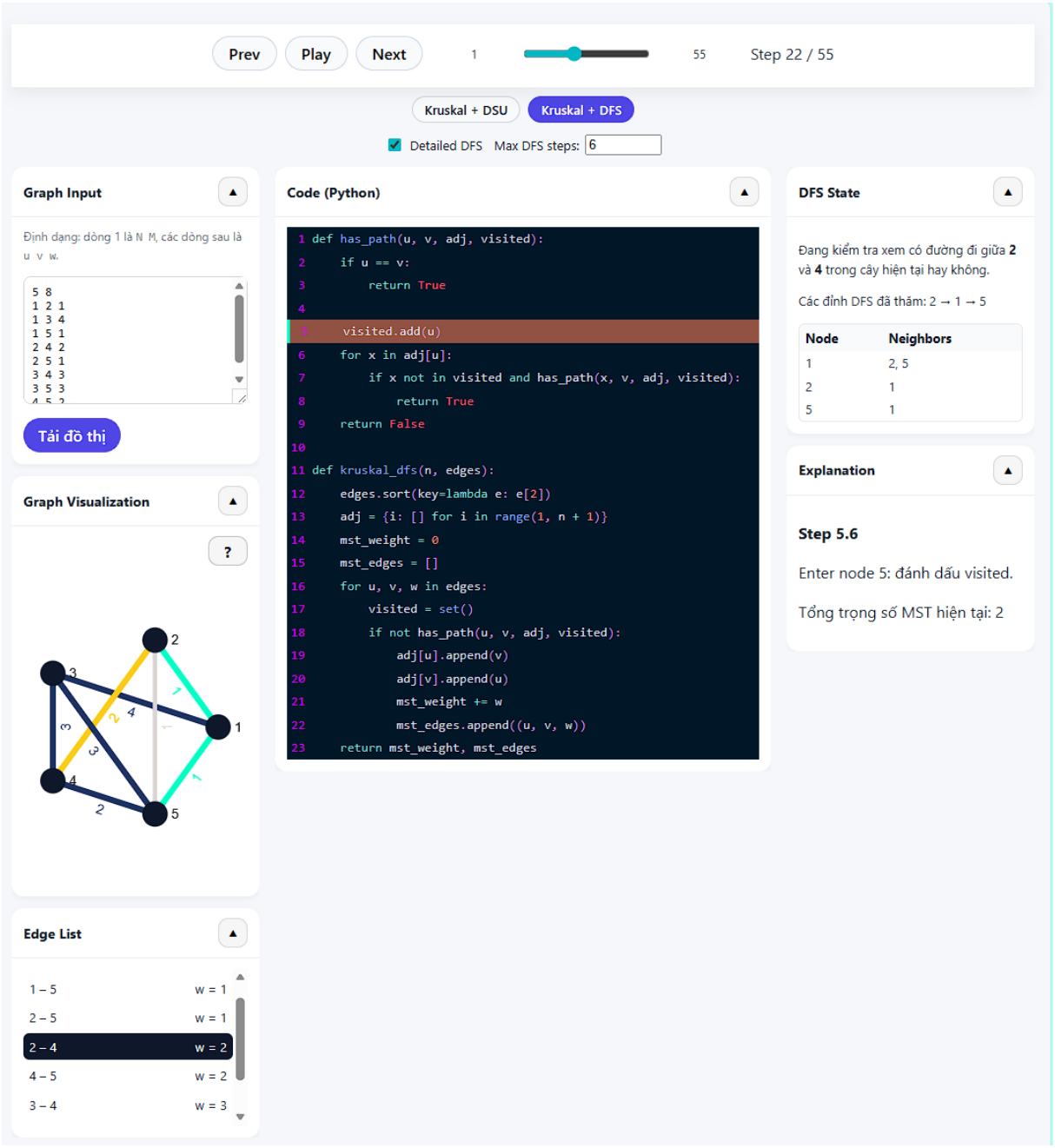

Figure 5: DFS mode showing cycle detection traversal

5.5 Responsive Layouts

Figure 6 và 7: Responsive design across different screen sizes.

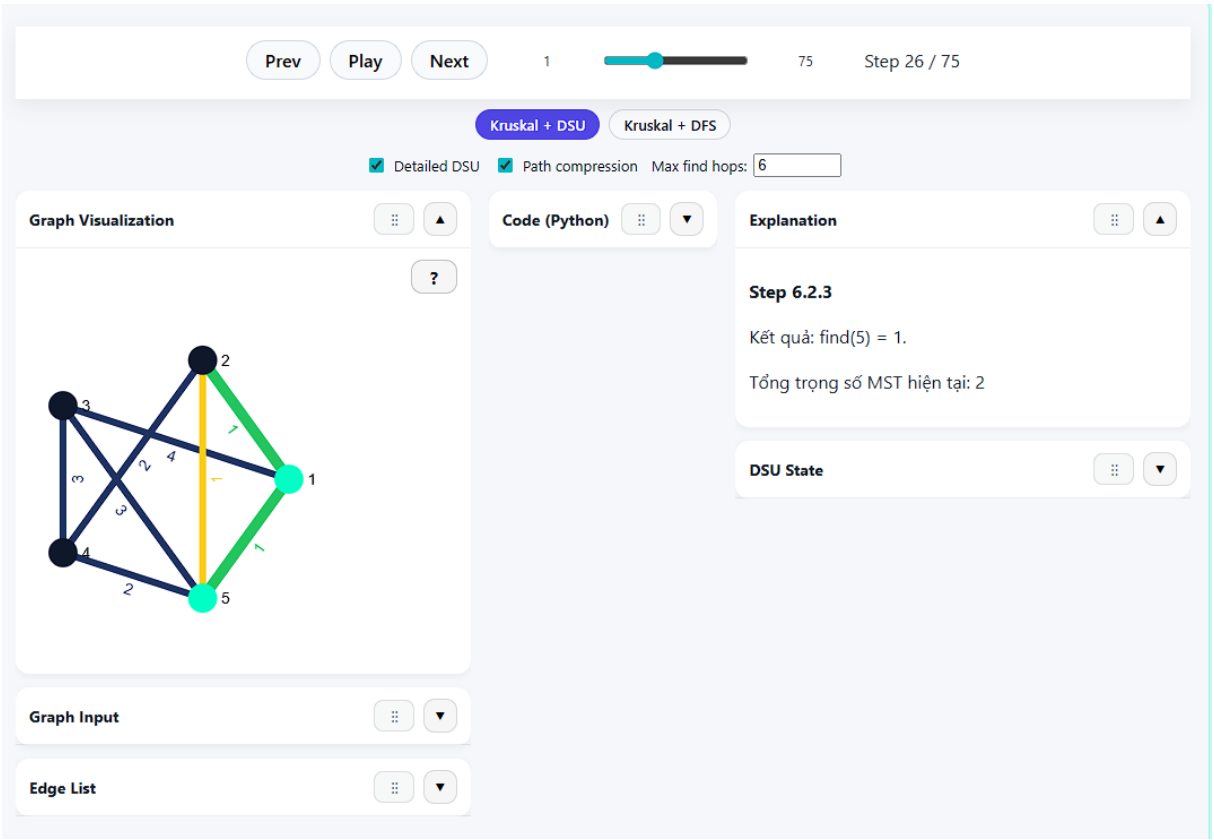

Figure 6: Desktop layout (3 columns) – (a)

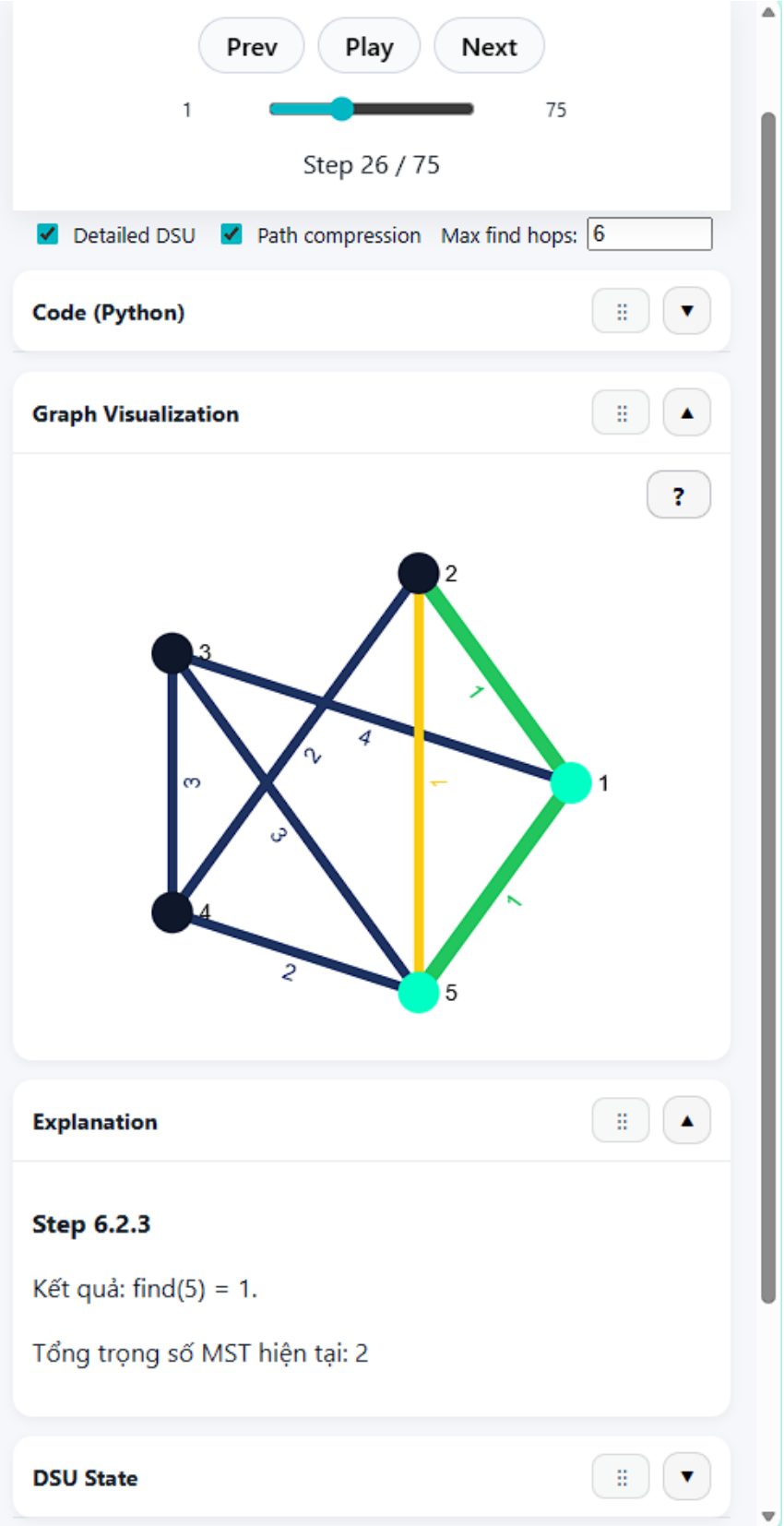

Figure 7: Mobile layout (1 column) – (b) (mobile version of 6)

6 Testing

6.1 Automated Testing

The project includes automated tests to verify algorithm correctness across all three implementations:

1. Kruskal + DSU with Path Compression
2. Kruskal + DSU without Path Compression
3. Kruskal + DFS

```
// DSU Implementation with Path Compression
class DSU_PC {
    parent: number[];
    rank: number[];

    find(u: number): number {
        if (this.parent[u] !== u) {
            this.parent[u] = this.find(this.parent[u]);
        }
        return this.parent[u];
    }
    // ...
}

// DSU Implementation without Path Compression
class DSU_NoPC {
    find(u: number): number {
        if (this.parent[u] !== u) {
            return this.find(this.parent[u]);
        }
        return this.parent[u];
    }
    // ...
}

// DFS-based Implementation
function kruskalDfs(n: number, edges: Edge[]): number {
    // Uses hasPath() for cycle detection
}

// Run all tests
function main() {
    const testFiles = fs.readdirSync(TEST_DIR)
        .filter(f => f.endsWith('.in'));

    for (const file of testFiles) {
        // Test all 3 versions
        // Compare with expected output
    }
}
```

Listing 11: Test script structure (test-all.ts)

6.2 Test Cases

Test cases are stored in the `src/test-cases/` directory with `.in` (input) and `.out` (expected output) file pairs.

Input format:

```
N M
u1 v1 w1
u2 v2 w2
...
```

Example (case1.in):

```
5 8
1 2 1
1 3 4
1 5 1
2 4 2
2 5 1
3 4 3
3 5 3
4 5 2
```

Running tests:

```
npx tsx scripts/test-all.ts
```

```
=== KẾT QUẢ TEST ===
Test          Expected    DSU+PC      DSU-NoPC    DFS         Result
------------------------------------------------------------------------
case1         146379      146379      146379      146379      ✅PASS
case2         5665102     5665102     5665102     5665102     ✅PASS
case3         139512619   139512619   139512619   139512619   ✅PASS
case4         372586      372586      372586      372586      ✅PASS
case5         26374598    26374598    26374598    26374598    ✅PASS
case_small    6           6           6           6           ✅PASS

🎉 TẤT CẢ TEST PASS!
```

Figure 8: Test execution showing all 3 versions passing

7 Deployment and Performance

7.1 Deployment on Vercel

The application is deployed on Vercel with automatic deployments on git push:

- **URL**: `https://kruskal-mst-visualizer.vercel.app/`
- **Build Command**: `npm run build`
- **Framework**: Vite
- **Node.js Version**: 18+

7.2 Performance Monitoring

Vercel Speed Insights is integrated for performance monitoring:

```
import { SpeedInsights } from "@vercel/speed-insights/react";

function App() {
    return (
        <div className="app-root">
            {/* ... app content ... */}
            <SpeedInsights />
        </div>
    );
}
```

Listing 12: Speed Insights integration

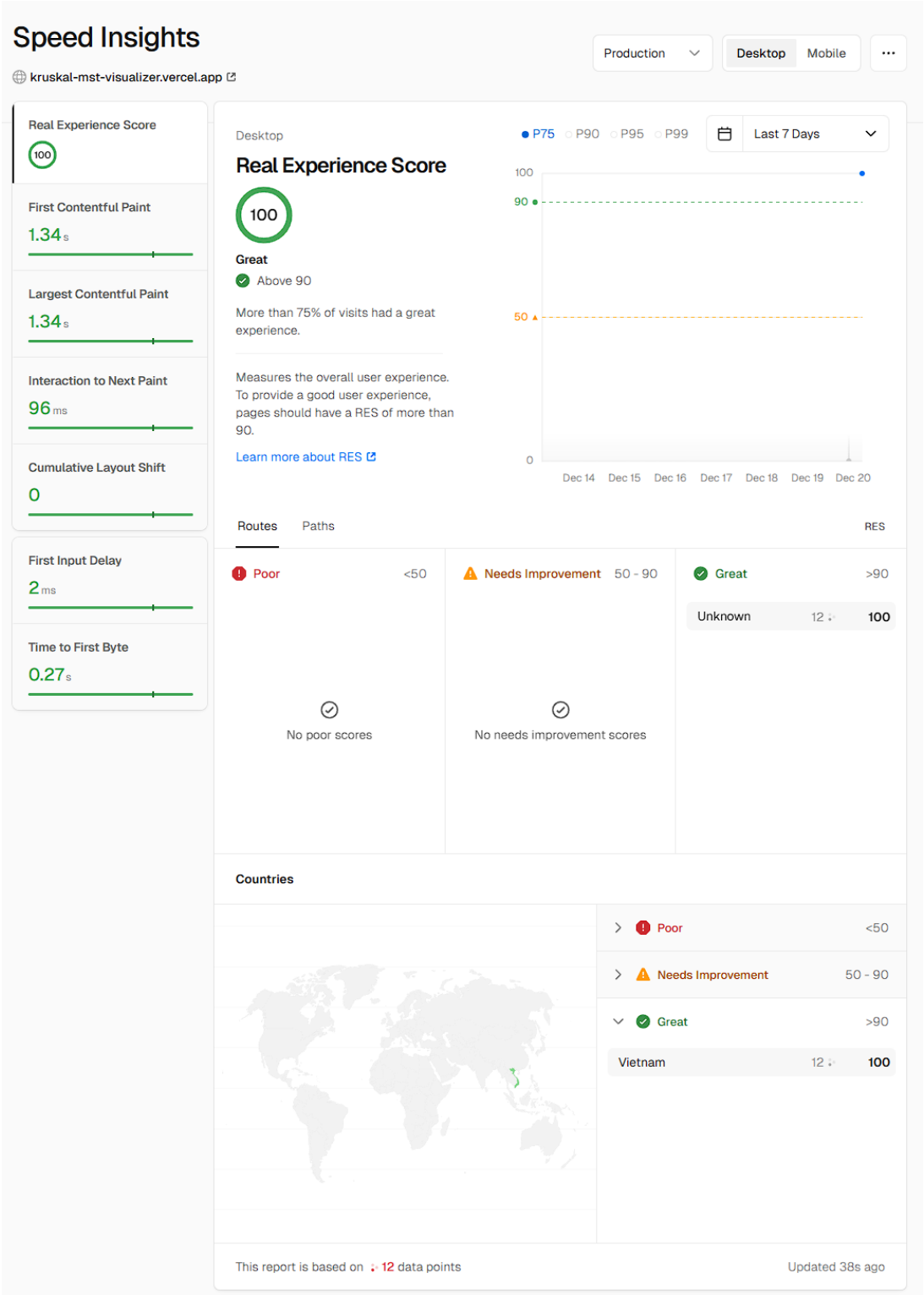

Figure 9: Vercel Speed Insights performance dashboard

8 Conclusion

8.1 Achievements

This project successfully achieved its objectives:

1. **Interactive Visualization**: Created a fully functional web application that visualizes Kruskal's algorithm step-by-step
2. **Multiple Implementation Approaches**: Implemented and visualized both DSU-based and DFS-based cycle detection methods, allowing users to compare different approaches
3. **Educational Value**: The detailed step-by-step execution with code highlighting and explanations helps users understand the algorithm's inner workings
4. **Modern Technology Stack**: Applied modern web development practices including React, TypeScript, and component-based architecture
5. **Responsive Design**: Created a user-friendly interface that works well on both desktop and mobile devices
6. **Customizable Experience**: Implemented draggable panels and collapsible sections for personalized workspace organization

8.2 Skills Developed

Through this project, the following practical programming skills were developed:

- React.js development with hooks and functional components
- TypeScript for type-safe application development
- Graph visualization using Sigma.js and Graphology
- Implementing complex algorithms with step-by-step state tracking
- Responsive web design with CSS
- Drag-and-drop interfaces with @dnd-kit
- Deployment and continuous integration with Vercel
- Writing automated tests for algorithm verification

8.3 Future Improvements

Potential areas for future development:

- Add more graph algorithms (Prim's, Dijkstra's, etc.)
- Support custom graph input via visual drawing
- Add animation speed control
- Implement step bookmarking and sharing
- Add multilingual support
- Create interactive tutorials for beginners

9 References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. React Documentation. `https://react.dev/`
3. TypeScript Handbook. `https://www.typescriptlang.org/docs/`
4. Sigma.js Documentation. `https://www.sigmajs.org/`
5. Graphology Documentation. `https://graphology.github.io/`
6. Vite Documentation. `https://vitejs.dev/`
7. @dnd-kit Documentation. `https://dndkit.com/`
8. Vercel Documentation. `https://vercel.com/docs`

A Source Code Repository

The complete source code is available on GitHub and can be accessed for review and contribution.

B How to Run Locally

```
# Clone the repository
git clone https://github.com/Relieq/kruskal-mst-visualizer
cd kruskal-mst-visualizer

# Install dependencies
npm install

# Start development server
npm run dev

# Build for production
npm run build

# Run tests
npx tsx scripts/test-all.ts
```