# Pathfinding Algorithm Visualizer Guide

## Project Overview

This project is a web-based **pathfinding algorithm visualizer** built with HTML, CSS, and vanilla JavaScript. It allows users to visualize how pathfinding algorithms (e.g., BFS, Dijkstra, A\*) find the shortest path between two points on a grid while navigating around obstacles. The project is designed to be challenging, requiring algorithmic implementation, performance optimization, and a polished user interface. It’s ideal for showcasing problem-solving, data structures, and front-end skills in your portfolio.

### Key Features

* **Interactive Grid**: A grid (20x20 to 50x50 cells) where users can set start/end points and draw walls.
* **Algorithms**: Implement at least BFS, Dijkstra, and A\* for pathfinding.
* **Visualization**: Animate the search process (visited nodes, final path) with stats (nodes visited, path length, time).
* **Controls**: Buttons for algorithm selection, speed control, reset, and random maze generation.
* **Polish**: Responsive design, undo functionality, diagonal movement toggle, and localStorage for saving grids.
* **Edge Cases**: Handle invalid inputs (e.g., no path exists) and optimize for large grids.

### Why It’s Challenging

Even with LLM assistance, you’ll need to:

* Debug complex algorithm logic (e.g., A\* heuristic correctness).
* Optimize performance to prevent browser lag on large grids.
* Sync animations with algorithm steps without freezing the UI.
* Handle edge cases and ensure a smooth, accessible user experience.

## Technical Requirements

* **HTML**: Structure the grid, controls, and stats display.
* **CSS**: Style the grid (using CSS Grid/Flexbox), animate cell states, ensure responsiveness.
* **JavaScript**: Implement algorithms, handle user interactions, manage animations, and optimize performance.
* **No external libraries**: Use vanilla JS for everything, including data structures (e.g., custom binary heap for priority queues).
* **Tools**: Browser dev tools for debugging, GitHub Pages for deployment.

## Implementation Steps

### 1. Project Setup

* **HTML Structure**:
  + Create a container for the grid (e.g., <table> or <div> with CSS Grid).
  + Add a sidebar for controls: buttons (start, reset, generate maze), dropdown for algorithm selection, slider for animation speed.
  + Include a stats section (nodes visited, path length, time taken).
* **File Structure**:
  + index.html: Main page.
  + styles.css: Grid and UI styling.
  + script.js: Logic for grid, algorithms, and interactions.
* **Initial Setup**:
  + Create a 20x20 grid by default (adjustable later).
  + Initialize start (top-left) and end (bottom-right) points.

### 2. Grid Implementation

* **HTML/CSS**:
  + Use a <table> or CSS Grid for the grid. Each cell is a <td> or <div> with classes like .cell, .wall, .start, .end, .visited, .path.
  + Style cells (e.g., 20px x 20px, border, background colors for states).
  + Use CSS transitions for smooth color changes (e.g., transition: background-color 0.1s).
  + Make the grid responsive with media queries (e.g., smaller cells on mobile).
* **JavaScript**:
  + Represent the grid as a 2D array of node objects: { x, y, isWall, isStart, isEnd, cost, parent }.
  + Add event listeners for user interactions:
    - Click to set start/end points.
    - Drag to draw/remove walls (mousedown, mousemove, mouseup).
    - Optional: Add keyboard support for accessibility.

### 3. Pathfinding Algorithms

Implement at least three algorithms. Each should return the visited nodes and the shortest path (if found).

#### Breadth-First Search (BFS)

* **Use Case**: Finds shortest path in unweighted grids.
* **Logic**:
  + Use a queue (JS array with push/shift).
  + Track visited nodes with a Set or array.
  + Store parent nodes to reconstruct the path.
* **Pseudo-code**:

queue = [startNode]

visited = Set()

while queue not empty:

node = queue.dequeue()

if node is endNode: reconstruct path

for each neighbor (up, down, left, right):

if neighbor not visited and not a wall:

queue.enqueue(neighbor)

visited.add(neighbor)

neighbor.parent = node

#### Dijkstra’s Algorithm

* **Use Case**: Handles weighted grids (if you add weights later).
* **Logic**:
  + Use a priority queue (implement a binary heap for efficiency).
  + Track distances and visited nodes.
* **Pseudo-code**:

pq = PriorityQueue(startNode, cost=0)

distances = { all nodes: Infinity, startNode: 0 }

while pq not empty:

node = pq.extractMin()

if node is endNode: reconstruct path

for each neighbor:

if not visited and newCost < distances[neighbor]:

distances[neighbor] = newCost

pq.update(neighbor, newCost)

neighbor.parent = node

#### A\* Algorithm

* **Use Case**: Optimizes search with a heuristic (faster than Dijkstra).
* **Logic**:
  + Use a priority queue with cost = g (distance from start) + h (heuristic estimate to end).
  + Heuristic: Manhattan distance (|x1-x2| + |y1-y2|) for 4-directional movement.
* **Pseudo-code**:

pq = PriorityQueue(startNode, g=0, h=manhattan(start, end))

gScores = { startNode: 0 }

fScores = { startNode: h }

while pq not empty:

node = pq.extractMin()

if node is endNode: reconstruct path

for each neighbor:

tentativeG = gScores[node] + cost(node, neighbor)

if tentativeG < gScores[neighbor]:

gScores[neighbor] = tentativeG

fScores[neighbor] = tentativeG + manhattan(neighbor, end)

pq.update(neighbor, fScores[neighbor])

neighbor.parent = node

#### Binary Heap (for Priority Queues)

* Implement a min-heap for Dijkstra and A\*:
  + Array-based, with methods: insert(node, priority), extractMin(), update(node, newPriority).
  + Maintain heap property via bubbleUp and bubbleDown.

### 4. Visualization and Animation

* **Logic**:
  + Store visited nodes and final path during algorithm execution.
  + Use requestAnimationFrame for smooth animations.
  + Update cell classes (e.g., .visited, .path) in batches.
* **Steps**:
  + After each algorithm step, add a delay (controlled by slider, e.g., 10-100ms).
  + Update DOM: Change cell classes for visited nodes, then the final path.
  + Avoid excessive DOM updates: Batch changes or use a document fragment.
* **Stats Display**:
  + Track nodes visited, path length, and execution time (use performance.now()).
  + Update stats in real-time or at the end.

### 5. Additional Features

* **Random Maze Generation**:
  + Use recursive division: Divide grid, add walls with gaps, repeat recursively.
  + Alternative: Prim’s algorithm for maze generation.
* **Undo Functionality**:
  + Store wall placements in a stack; pop to undo.
* **Diagonal Movement**:
  + Add toggle to allow 8-directional movement (update algorithms accordingly).
* **localStorage**:
  + Save grid state (walls, start/end) on button click; load on page refresh.
* **Edge Cases**:
  + Check if start/end are blocked or no path exists (display error message).
  + Optimize for large grids (e.g., 50x50) by minimizing DOM updates and using efficient data structures.

### 6. Optimization and Debugging

* **Performance**:
  + Use a single requestAnimationFrame loop for animations.
  + Avoid nested loops in DOM updates; use CSS for transitions.
  + Profile with Chrome DevTools to find bottlenecks.
* **Debugging**:
  + Test algorithms with small grids (e.g., 5x5) to verify correctness.
  + Log node visits to console for debugging.
  + Handle edge cases (e.g., clicking same cell for start/end).

### 7. Deployment

* Deploy to GitHub Pages for a live demo.
* Test on mobile devices to ensure responsiveness.
* Add a README with setup instructions and screenshots.

## Technical Tips

* **Grid Representation**: Use a 2D array for simplicity, but consider a 1D array for faster lookups (index = row \* cols + col).
* **Event Handling**:
  + Use event delegation (e.g., attach listener to grid container) to reduce memory usage.
  + Debounce rapid clicks/drags to prevent lag.
* **Algorithm Optimization**:
  + For A\*, ensure heuristic is admissible (never overestimates) to guarantee shortest path.
  + Cache neighbor calculations to avoid redundant checks.
* **Accessibility**:
  + Add aria-label to grid cells and buttons.
  + Support keyboard navigation (e.g., arrow keys to move start/end points).
* **Testing**:
  + Test with extreme cases: empty grid, fully blocked, single-cell path.
  + Use Jest or console-based tests for algorithm logic.

## Resources

* **Algorithms**: Study BFS, Dijkstra, A\* on GeeksforGeeks or Wikipedia (focus on pseudocode).
* **Performance**: MDN docs for requestAnimationFrame, Chrome DevTools for profiling.
* **Inspiration**: Search “pathfinding visualizer demo” for ideas (avoid copying code).
* **CSS Grid**: CSS-Tricks guide for responsive layouts.
* **Accessibility**: WebAIM for ARIA and keyboard navigation tips.

## Challenges to Anticipate

* **Algorithm Bugs**: A\* may take wrong paths if heuristic is miscalculated.
* **Performance**: Large grids (50x50) may lag without optimization.
* **UI Sync**: Animations may desync from algorithm steps if delays are inconsistent.
* **Edge Cases**: Handling invalid inputs (e.g., start=end) requires careful validation.

## Next Steps

1. Start with a small 10x10 grid and BFS implementation.
2. Add visualization and controls incrementally.
3. Test and debug each algorithm before moving to the next.
4. Optimize performance after basic functionality works.
5. Polish UI and deploy to GitHub Pages.

This project will take 10-20 hours, depending on your experience. It’s a strong portfolio piece that demonstrates algorithmic thinking, UI design, and performance optimization. Good luck, and enjoy building!