PATH FINDING VISULAISER

Presented to:

Dr. Sara Khalil

Prepared by:

مريم عادل عبدالعظيم عبدالله

نادين أحمد محمد الهادي

نادين هيثم فتح الله عبدالحفيظ

هنا أحمد سعيد رشاد

Overview:

This project is a dynamic and interactive Pathfinding Visualizer built using React and TypeScript. It is designed to help users understand how pathfinding algorithms work by visualizing them in real-time on a 2D grid.
Users can:

- Place a start and end node
- Add or remove walls
- Trigger Dijkstra's or A* algorithm to see how they explore the grid and find the shortest path

The visualizer supports smooth animations, grid interaction, and live comparisons between the two algorithms.

🧠 Algorithms Implemented:

- Dijkstra's Algorithm (uninformed, guarantees shortest path)
- A* (informed, uses heuristics for faster computation)

Both algorithms were optimized to minimize redundant operations, and visual feedback shows their order of node visits and final path.

🔗 Project link:

https://path-finding-visualizer-qc53.vercel.app/

Project GUI:

The Pathfinding Visualizer includes a clean and interactive GUI developed using React.js. It enables users to explore various pathfinding algorithms visually on a grid-based system. The application is web-deployed for ease of access and platform independence.

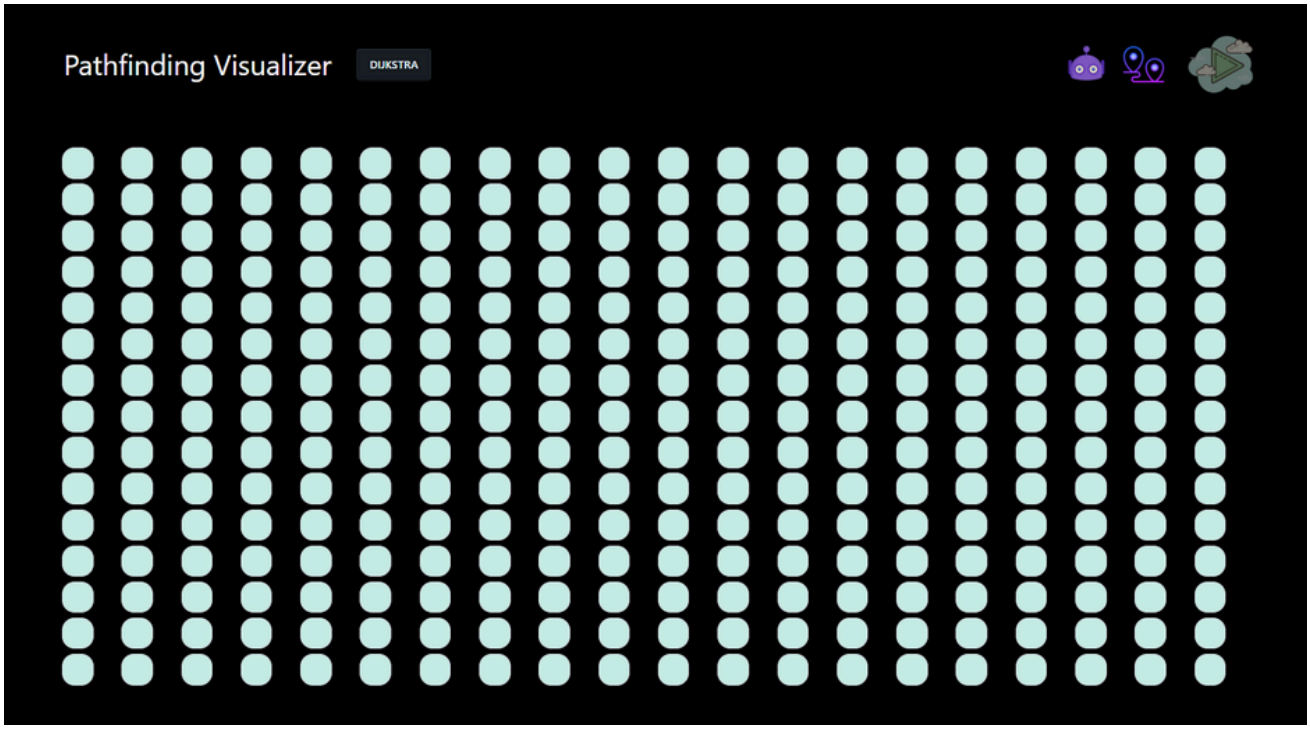
Interface Features:

- Algorithm Selection: Users can choose from algorithms like Dijkstra via a clear button interface.
- Interactive Grid: Each cell represents a node that can be a wall, start, end, or part of a path.
- Toolbar Icons: Includes options for running the algorithm, generating mazes, and resetting the grid.

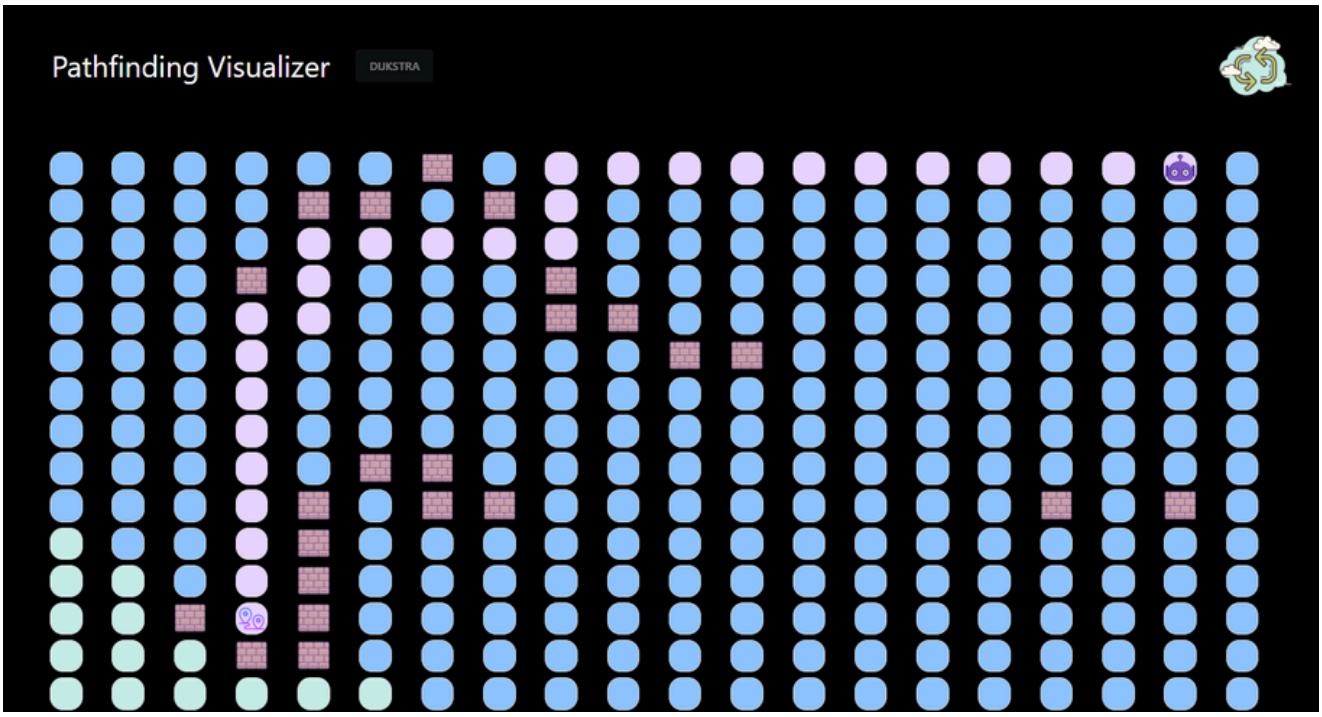
Application Rules:

- Restarting During Animation: If the grid is mid-animation, pressing the "Restart" button will immediately reset it.
- Preserving Start/End Points: When restarting after an algorithm has completed, the start and end nodes are preserved. This allows users to compare the performance and behavior of different algorithms under the same conditions.
- Wall Restrictions: Users are not allowed to place walls on top of the start or end nodes to prevent unintentional errors or invalid states during execution.

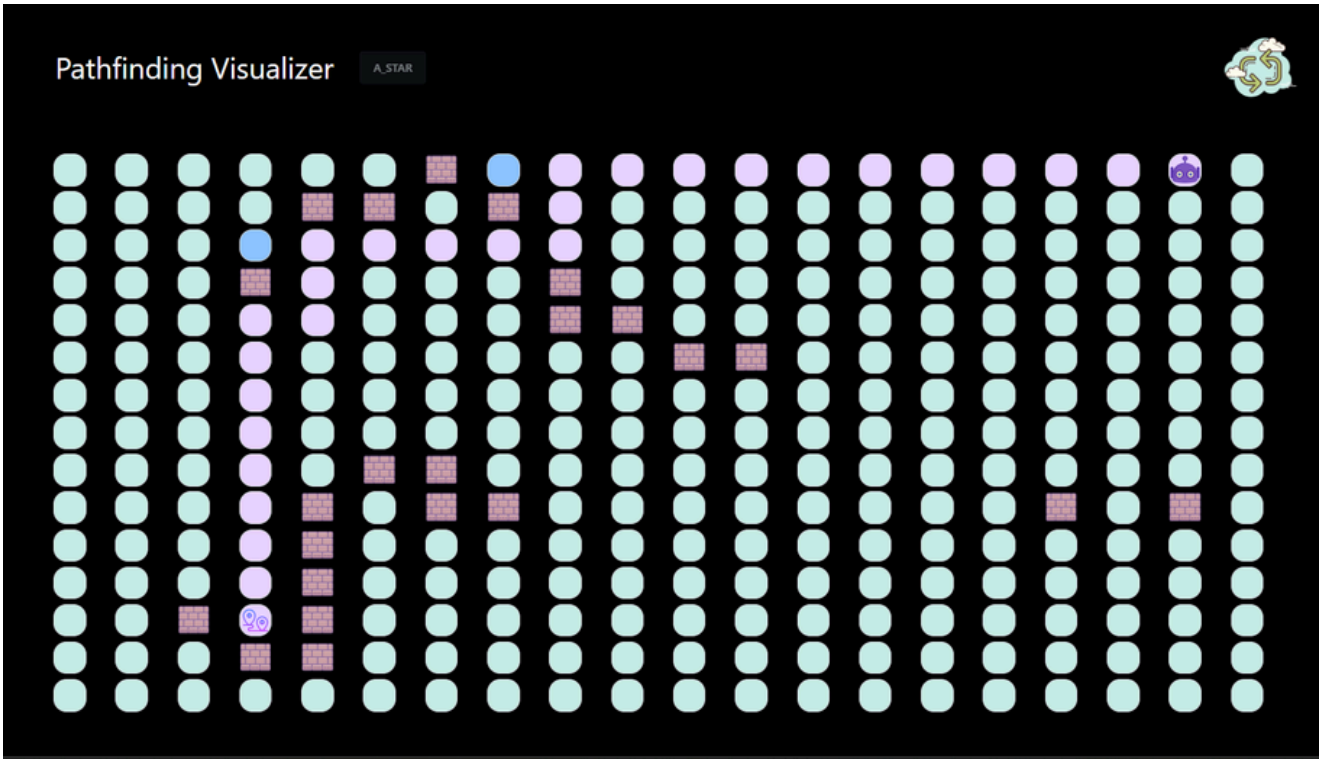
These rules are intended to balance functionality with development simplicity, making the visualizer a practical tool for testing and comparing pathfinding algorithms quickly.



Project in action:



Using Dijkstra's Algorithm.



Using A* algorithm.

Dijkstra's Algorithm:

Dijkstra's Algorithm is a popular algorithm used to find the shortest path from a starting node to all other nodes in a weighted graph with non-negative edge weights. It works by repeatedly selecting the closest unvisited node, updating the shortest distances to its neighbors, and marking nodes as visited once their shortest distance is finalized.

- **Pseudocode:**

```
FUNCTION getNeighbors(node, grid, maxRow, maxCol)
    directions ← [(0,1), (1,0), (0,-1), (-1,0)]
    neighbors ← empty list

    FOR each (dx, dy) in directions
        newRow ← node.row + dx
        newCol ← node.col + dy
        id ← "node-newRow-newCol"
        IF newRow and newCol are within bounds AND grid[id] exists AND NOT a wall
            ADD grid[id] to neighbors

    RETURN neighbors


FUNCTION dijkstra(startId, endId, wallPositions, maxRow, maxCol)
    grid ← empty map<NodeId, GridNode>
    visitedOrder ← empty list

    // Initialize grid nodes
    FOR row FROM 0 TO maxRow - 1
        FOR col FROM 0 TO maxCol - 1
            id ← "node-row-col"
            grid[id] ← {
                id: id,
                row: row,
                col: col,
                distance: ∞,
                isWall: (id ∈ wallPositions),
                previousNode: null
            }

    grid[startId].distance ← 0

    minHeap ← empty min-priority queue
    ADD grid[startId] TO minHeap

    visitedSet ← empty set

    // Main loop
    WHILE minHeap IS NOT empty
        current ← minHeap.dequeue()  // node with smallest distance
        IF current.id IN visitedSet, CONTINUE
        ADD current.id TO visitedSet

        IF current.isWall, CONTINUE
        ADD current.id TO visitedOrder

        IF current.id == endId, BREAK

        neighbors ← getNeighbors(current, grid, maxRow, maxCol)
        FOR each neighbor IN neighbors
            alt ← current.distance + 1
            IF alt < neighbor.distance
                neighbor.distance ← alt
                neighbor.previousNode ← current.id
                ADD neighbor TO minHeap

    // Reconstruct path
    shortestPath ← empty list
    currentId ← endId

    WHILE currentId IS NOT null AND grid[currentId].previousNode IS NOT null
        PREPEND currentId TO shortestPath
        currentId ← grid[currentId].previousNode

    IF currentId == startId
        PREPEND startId TO shortestPath

    RETURN (visitedOrder, shortestPath)
```

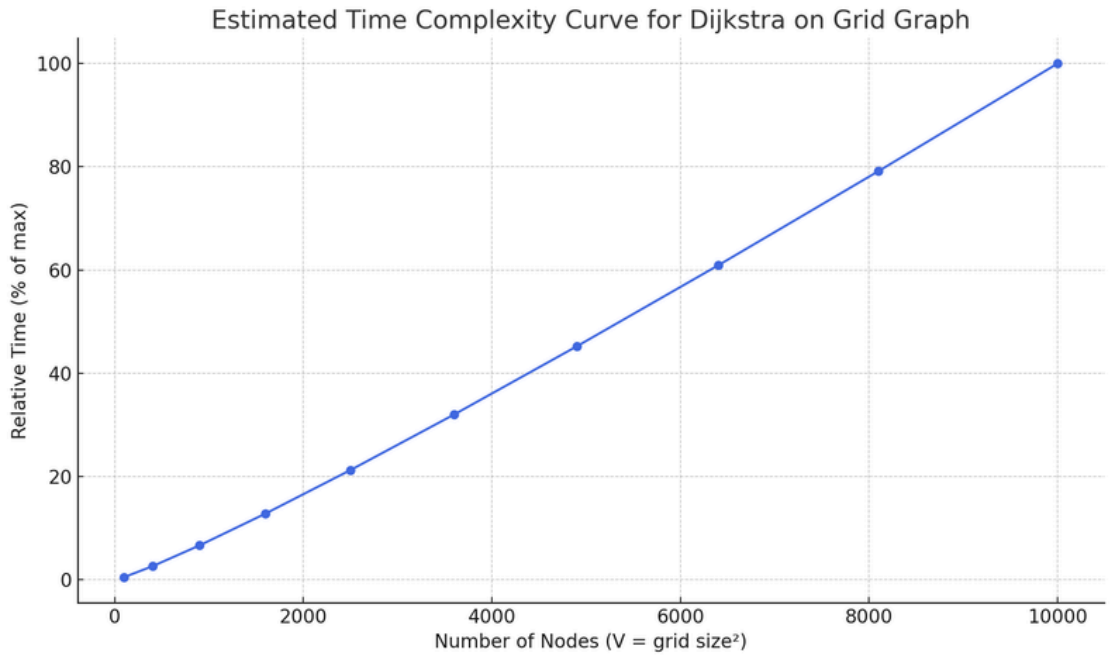
- **Time complexity analysis:**

Let:

- R = number of rows
- C = number of columns
- V = R × C = total number of nodes
- E ≈ 4V = total number of edges in a grid (each node connects to up to 4 neighbors)
- We are using a min-heap (priority queue) for selecting the next node

Operation	Frequency	Time per Operation	Total Time
Grid initialization	V times	O(1)	O(V)
Add all nodes to priority queue	V times (max)	O(log V)	O(V log V)
Dequeue node with smallest distance	Up to V times	O(log V)	O(V log V)
Neighbor lookup (getNeighbors)	Up to 4 per node	O(1)	O(E)
Edge relaxation	Up to E times	O(log V) (enqueue)	O(E log V)
Path reconstruction	Up to V steps	O(1)	O(V)

Total time complexity: O((V + E) log V)
In grid graphs where E ≈ 4V: O((V + 4V) log V) = O(5V log V) = O(V log V)



A* Algorithm:

A* (A-Star) is an informed graph search algorithm that finds the shortest path from a start node to a goal node while using heuristics to guide its search efficiently. Unlike other traversal techniques, it has "brains". It is widely used in pathfinding and AI due to its balance of performance and optimality.

• How it works:

A* combines the strengths of Dijkstra's algorithm and greedy best-first search. What A* Search Algorithm does is that at each step it picks the node according to a value 'f' which is a parameter equal to the sum of two other parameters 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.
We define 'g' and 'h' as simply as possible below

- g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
- h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess.

By prioritizing nodes with the lowest f(n), A* explores paths that are both efficient and promising.

• Approximate heuristic calculations:

1. **Manhattan Distance:**
- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively.

2. **Diagonal Distance:**
- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively.

3. **Euclidean Distance:**
- it is nothing but the distance between the current cell and the goal cell using the distance formula.

In our implementation we used the Manhattan Distance as it only moves in four directions so it is more suitable.

- **Pseudocode:**

```
function A_STAR(startNode, endNode, wallPositions, maxRow, maxCol):
    initialize grid with nodes
    for each node in grid:
        node.g = ∞
        node.f = ∞
        node.h = ∞
        node.previousNode = null
        node.isWall = (node.id in wallPositions)

    startNode.g = 0
    startNode.h = heuristic(startNode, endNode)
    startNode.f = startNode.g + startNode.h

    openSet = MinPriorityQueue()
    openSet.enqueue(startNode, priority=startNode.f)

    visitedOrder = []

    while openSet is not empty:
        current = openSet.dequeue()

        if current is endNode:
            break

        if current.isWall:
            continue

        visitedOrder.append(current)

        for each neighbor of current:
            if neighbor is a wall:
                continue

            tentativeG = current.g + 1
            if tentativeG < neighbor.g:
                neighbor.previousNode = current
                neighbor.g = tentativeG
                neighbor.h = heuristic(neighbor, endNode)
                neighbor.f = neighbor.g + neighbor.h
                openSet.enqueue(neighbor, priority=neighbor.f)

    shortestPath = reconstructPath(endNode)
    return visitedOrder, shortestPath
```

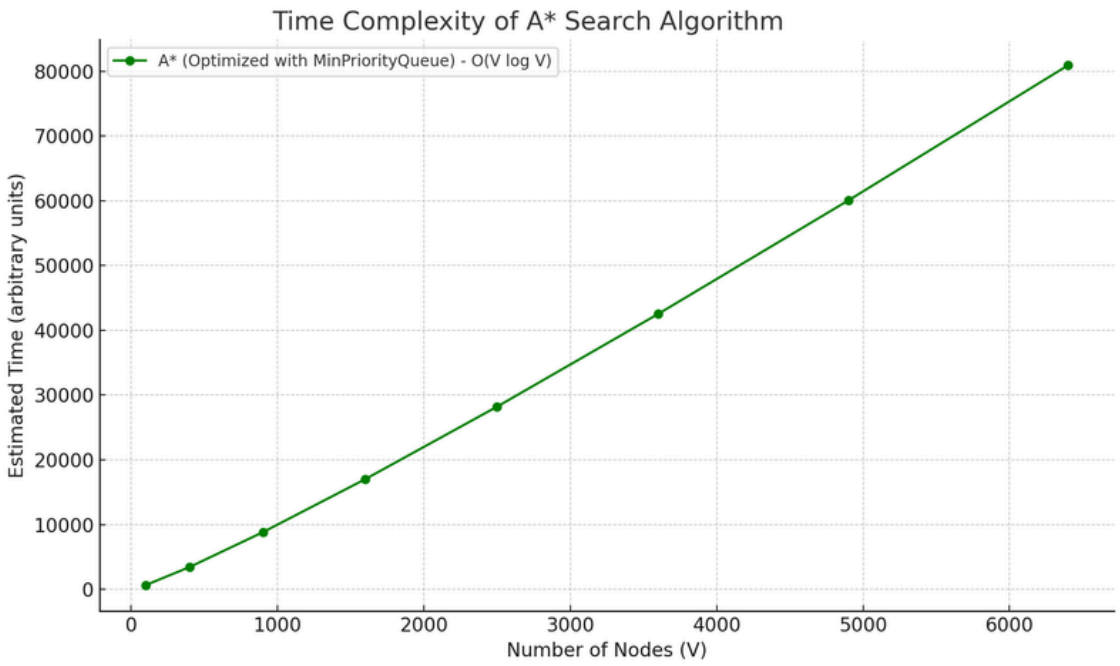
- **Time complexity analysis:**

Let:

- V be the total number of nodes (vertices) = maxRow * maxCol
- E be the number of edges ≈ 4V (each node has up to 4 neighbors in a grid)

Operation	Frequency	Time per Operation	Total Time
Grid initialization	V	O(1)	O(V)
Dequeue from MinPriorityQueue	≤ V	O(log V)	O(V log V)
Processing neighbors (up to 4 per node)	≤ 4V	O(log V)	O(V log V)
Update node values	≤ 4V	O(1)	O(V)
Reconstructing shortest path	≤ V	O(1)	O(V)

Total time complexity: O(VlogV)



Discussion of Results:

- Both algorithms guarantee finding the shortest path if one exists.
- Both maintain a list of visited nodes and track the shortest known distance from the start node to every other node.
- The grid is initialized similarly for both algorithms, with each node assigned attributes like distance, wall status, and previous node tracking.
- Both explore neighbors in the four cardinal directions (up, down, left, right).

Observations from Experimentation:

- A* consistently visits fewer nodes to reach the target, especially in large and open grids.
- Dijkstra's explores many unnecessary nodes because it has no concept of direction.
- In scenarios with dense obstacles, both algorithms show smilar performance, but A* still retains a slight edge due to its guidance mechanism.
- The path length found by both algorithms is the same in all cases—both are optimal.

Conclusion:

- If you want guaranteed optimal results and have no heuristic, use Dijkstra's.
- If you have a good heuristic (like Manhattan distance in grids) and want better speed, A* is the superior choice.
- In practical applications (like pathfinding in games or maps), A* is typically preferred due to its efficiency.