INTERNATIONAL UNIVERSITY
School of Computer Science and Engineering
(Sem 2, 2024 - 2025)

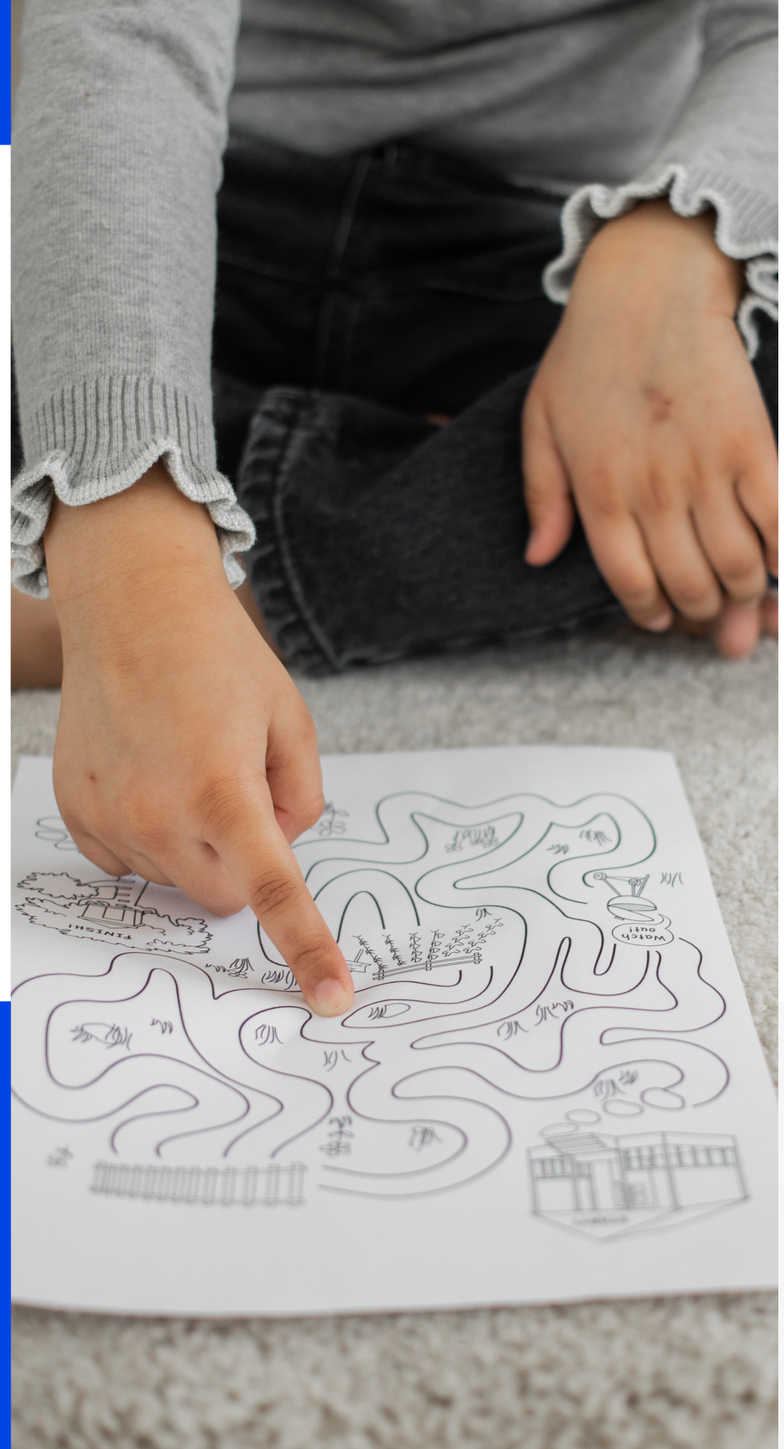
STUDENT PROJECT

PATHFINDING VISUALIZER

Algorithms & Data Structure

Lê Hưng - ITCSIU22271 (Gr: 0202)

Trần Đăng Nhất - ITITIU22115 (Gr: 0201)



CONTENT

INTRODUCTION

In this part, we explain about objective, programming languages, algorithm & data structure applied and some highlighted features.

OBJECTIVE

Clarify Algorithm Behavior

Maze Generation

LANGUAGES

React

React for building a modular and reactive user interface

TypeScript

TypeScript for type safety and better code maintainability

tailwindcss

Tailwind CSS for fast and responsive styling

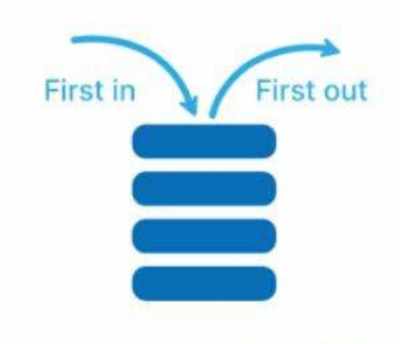
DEVELOPMENT ENVIRONEMT

Visual Studio Code

GitHub

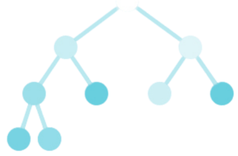
ALGORITHMS AND DATA STRUCTURES

Generated Maze



Stack

Queue



Min-heap

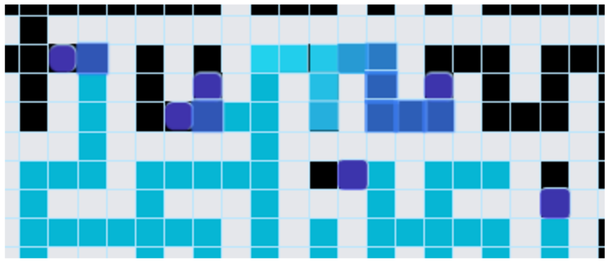
- Depth First Search
- Breadth First Search
- Dijkstra's Search
- A* Search

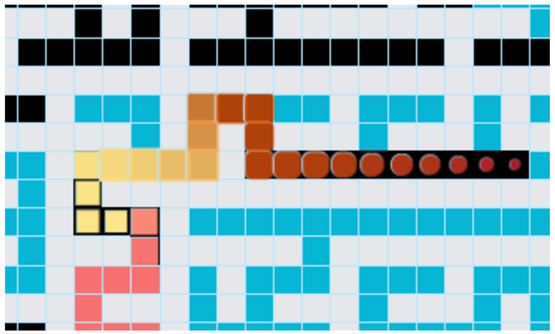
SOLVED

SHORTEST PATH

HIGHLIGHT

01 Real-Time Animation





isWall()

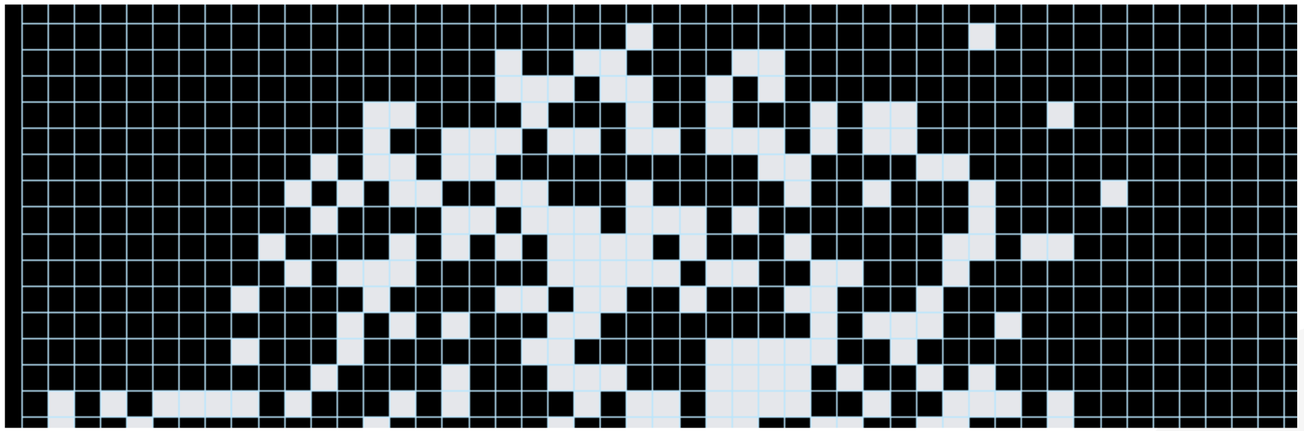
isTraversed()

Optimal path

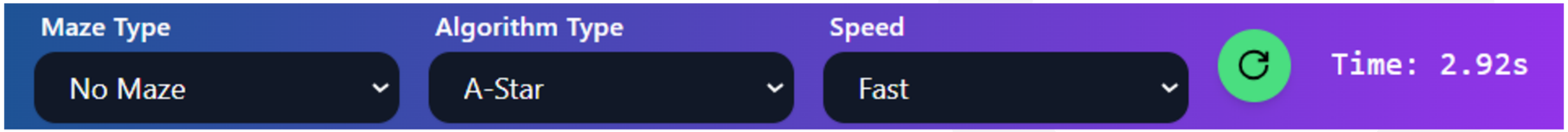
Animation traversing path

Animation finding Optimal path

02 Interactive User Interface



03 Timer for each run

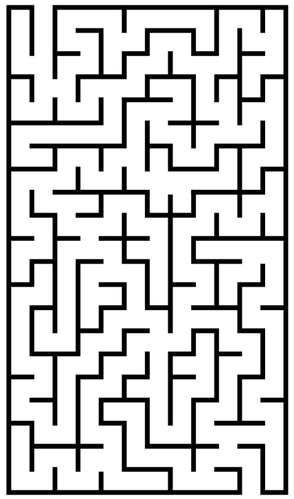


UI DESIGN

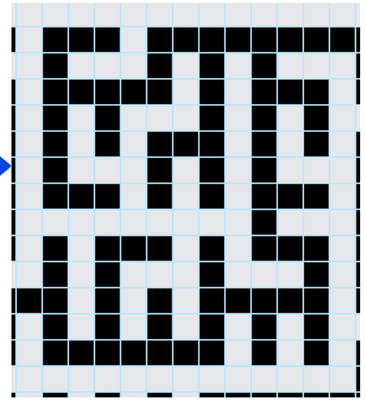
In this part, we introduce about the UI design for visualizing how the path is traversed in different algorithms and how the maze is generated.

ALGORITHMS AND MAZE TYPES

Binary Tree



Recursive Division



- Depth First Search
- Breadth First Search
- Dijkstra's Search
- A* Search

SOLVED

SHORTEST PATH

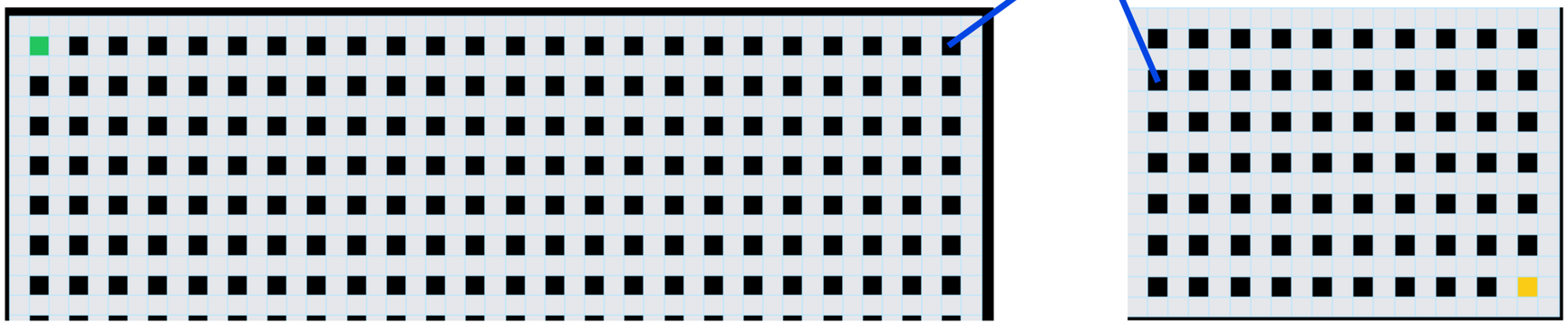
BINARY TREE MAZE

The Binary Tree algorithm is a simple and efficient maze generation technique. Here is how it works in our project

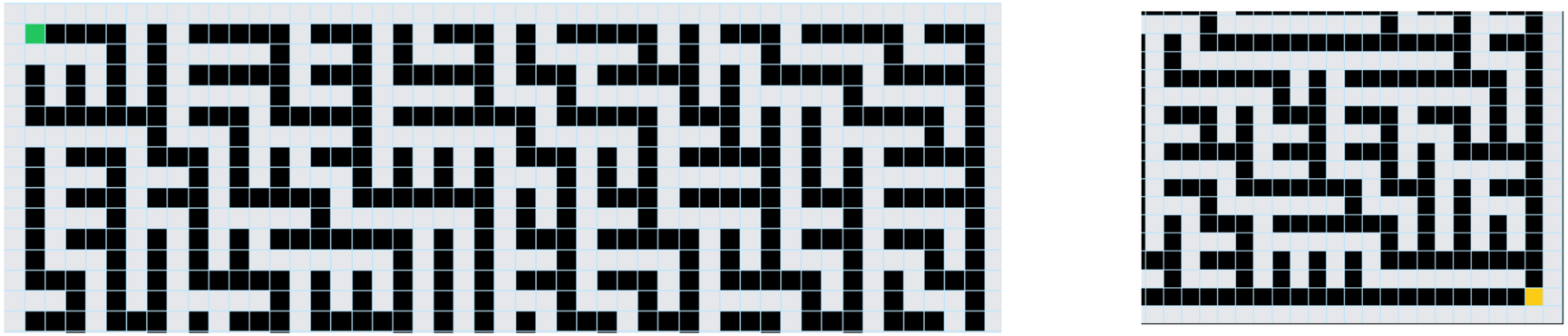
Starting point

Destination

Step 1 **Creating a grid pattern**



Step 2 **Removing the wall by rule**



- If it is the bottom-right corner, skip it (to keep an exit open).
- If it is on the last row, remove the wall to the right (destroyWall direction 1).
- If it is on the last column, remove the wall below (destroyWall direction 0).
- Otherwise, randomly remove either the right wall or the bottom wall to create a path, forming an open maze.

CREATING A GRID PATTERN

LINE OF CODE

```ts
export const createWall = (
  startTile: TileType,
  endTile: TileType,
  speed: SpeedType
) => {
  const delay = 6 * SPEEDS.find((s) => s.value === speed)!.value - 1;

  for (let row = 0; row < MAX_ROWS; row++) {
    setTimeout(() => {
      for (let col = 0; col < MAX_COLS; col++) {
        if (row % 2 === 0 || col % 2 === 0) {
          if (
            !isRowColEqual(row, col, startTile) &&
            !isRowColEqual(row, col, endTile)
          ) {
            setTimeout(() => {
              document.getElementById(
                `${row}-${col}`
              )!.className = `${WALL_TILE_STYLE} animate-wall`;
            }, delay * col);
          }
        }
      }
    }, delay * (MAX_ROWS / 2) * row);
  }
};
```

createWall.ts

REMOVING THE WALL BY RULE

LINE OF CODE

```
for (let r = 1; r < MAX_ROWS; r += 2) {
  // Iterate through odd rows starting from 1
  for (let c = 1; c < MAX_COLS; c += 2) {
    // Iterate through odd columns starting from 1
    if (r === MAX_ROWS - 2 && c === MAX_COLS - 2) {
      // Skip the bottom-right corner
      continue;
    } else if (r === MAX_ROWS - 2) {
      // If it's the last row, destroy a wall to the right
      await destroyWall(grid, r, c, 1, speed);
    } else if (c === MAX_COLS - 2) {
      // If it's the last column, destroy a wall below
      await destroyWall(grid, r, c, 0, speed);
    } else {
      // Otherwise, randomly destroy a wall to the right or below
      await destroyWall(grid, r, c, getRandInt(0, 2), speed);
    }
  }
}
setIsDisabled(false); // Re-enable the UI
};
```

binaryTree.ts

RECURSIVE DIVISION MAZE

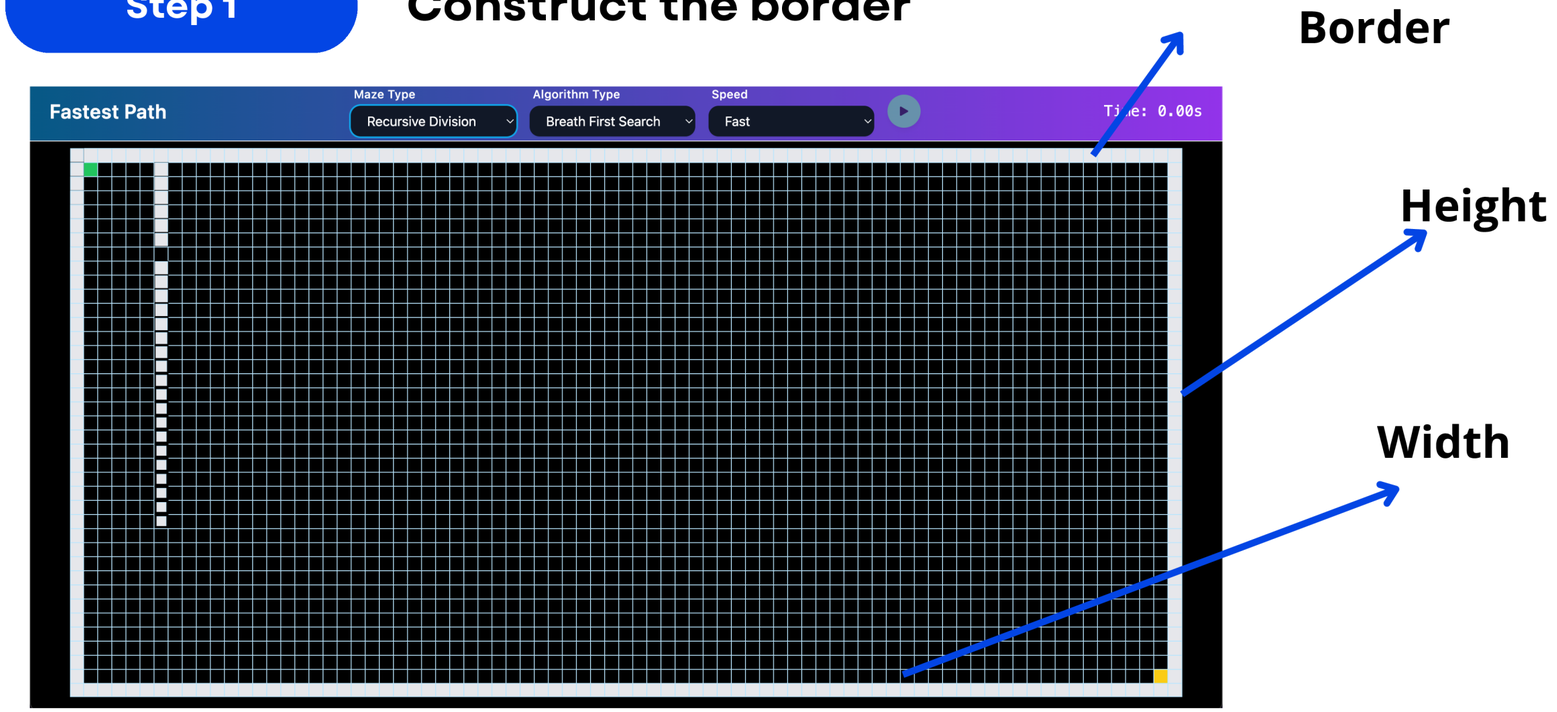
The Maze will be divied in 2 scenarios: Vertically and Horizontally

Starting point

Destination

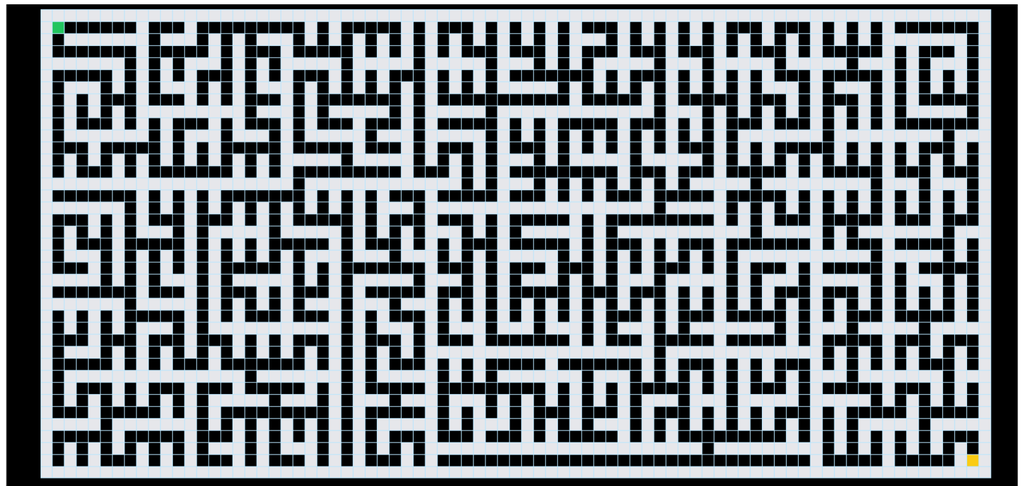
Step 1

Construct the border



Step 2

Starting the recursive division process to construct full maze



HOW IT WORKS?

The Maze will be divied in 2 scenarios: Vertically and Horizontally

Starting point

Destination

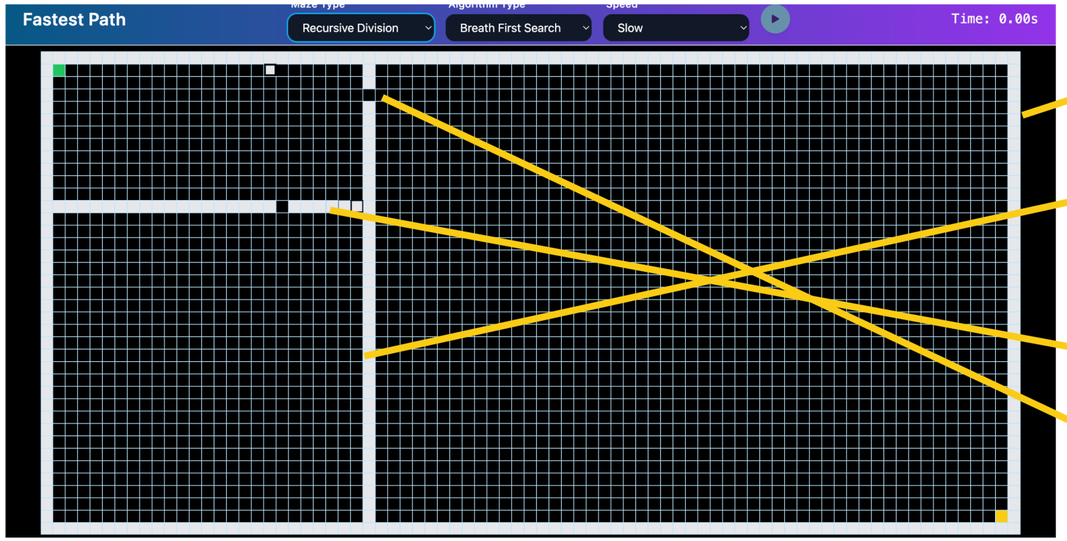
THE RECURSIVE PROCESS

Base case

If the current section of the grid is too small (height ≤ 1 or width ≤ 1), the recursion stops.

Division Choice

- If the section is taller than it is wide, it divides horizontally using **horizontalDivision**.
- Otherwise, it divides vertically using **verticalDivision**.



1) Height < width

2) devide vertically, now height > width (focus on the left side)

3) devide horizontally

Random Picks a Tiles for passing

RECURSIVE DIVISION

LINE OF CODE

```
if (height <= 1 || width <= 1) {
  return; // Base case: if the section is too small, stop recursion
}

if (height > width) {
  await horizontalDivision({
    // Divide horizontally if height is greater than width
    grid,
    startTile,
    endTile,
    row,
    col,
    height,
    width,
    setIsDisabled,
    speed,
  });
} else {
  await verticalDivision({
    // Divide vertically if width is greater than or equal to height
    grid,
    startTile,
```

recursiveDivision.ts

HORIZONTAL DIVISION

LINE OF CODE

Step 1. Construct the wall and random for the passage tile

```
6   export async function horizontalDivision({
26  }) {
27    const makeWallAt = row + getRandInt(0, height - 1) * 2 + 1; // Determine the row to place the wall
28    const makePassageAt = col + getRandInt(0, width) * 2; // Determine the column to leave a passage
29
30    for (let i = 0; i < 2 * width - 1; i += 1) {
31      // Create the horizontal wall
32      if (makePassageAt !== col + i) {
33        if (
34          !isEqual(grid[makeWallAt][col + i], startTile) && // Check if the current tile is not the start tile
35          !isEqual(grid[makeWallAt][col + i], endTile) // Check if the current tile is not the end tile
36        ) {
37          grid[makeWallAt][col + i].isWall = true; // Set the current tile as a wall
38
39          document.getElementById(
40            `${makeWallAt}-${col + i}`
41          )!.className = `${WALL_TILE_STYLE} animate-wall`; // Add wall style and animation
42          await sleep(10 * SPEEDS.find((s) => s.value === speed)!.value - 5); // Wait for animation
```

Step 2. call the recursiveDivision again for the the sections above and below the wall

```
await recursiveDivision({
  grid,
  startTile,
  endTile,
  row,
  col,
  height: (makeWallAt - row + 1) / 2,
  width,
  setIsDisabled,
  speed,
});
```

```
  });
  await recursiveDivision({
    grid,
    startTile,
    endTile,
    row: makeWallAt + 1,
    col,
    height: height - (makeWallAt - row + 1) / 2,
    width,
    setIsDisabled,
    speed,
  });
}
```

horizontalDivision.ts

VERTICAL DIVISION

LINE OF CODE

Step 1. Construct the wall and random for the passage tile

```
export async function verticalDivision({
  const makeWallAt = col + getRandInt(0, width - 1) * 2 + 1; // Determine the column to place the wall
  const makePassageAt = row + getRandInt(0, height) * 2; // Determine the row to leave a passage

  for (let i = 0; i < 2 * height - 1; i += 1) {
    // Create the vertical wall
    if (makePassageAt !== row + i) {
      if (
        !isEqual(grid[row + i][makeWallAt], startTile) && // Check if the current tile is not the start tile
        !isEqual(grid[row + i][makeWallAt], endTile) // Check if the current tile is not the end tile
      ) {
        grid[row + i][makeWallAt].isWall = true; // Set the current tile as a wall

        document.getElementById(
          `${row + i}-${makeWallAt}`
        )!.className = `${WALL_TILE_STYLE} animate-wall`; // Add wall style and animation
        await sleep(10 * SPEEDS.find((s) => s.value === speed)!.value - 5); // Wait for animation
      }
```

Step 2. call the recursiveDivision again for the the sections above and below the wall

```
// Recursively divide the sections to the left and right of the wa
await recursiveDivision({
  grid,
  startTile,
  endTile,
  row,
  col,
  height,
  width: (makeWallAt - col + 1) / 2,
  setIsDisabled,
  speed,
});
```

```
await recursiveDivision({
  grid,
  startTile,
  endTile,
  row,
  col: makeWallAt + 1,
  height,
  width: width - (makeWallAt - col + 1) / 2,
  setIsDisabled,
  speed,
});
```

verticalDivision.ts

ALGORITHMS

In this part, we introduce about the implementations of Data Structures and Algorithms to solve the maze path finding.

DEPTH FIRST SEARCH (DFS)

```
export const dfs = (grid: GridType, startTile: TileType, endTile: TileType) => {
  const traversedTiles = []; // Initialize an array to store traversed tiles
  const base = grid[startTile.row][startTile.col]; // Get the start tile from the grid
  base.distance = 0; // Set the distance of the start tile to 0
  base.isTraversed = true; // Mark the start tile as traversed
  const untraversedTiles = [base]; // Initialize the stack with the start tile

  while (untraversedTiles.length > 0) {
    // Continue while there are untraversed tiles
    const currentTile = untraversedTiles.pop(); // Get the last tile from the stack
```

Main while loop

While the stack is not empty: Pop the last tile added (DFS goes deep first).

```
if (currentTile) {
  // If the current tile is valid
  if (currentTile.isWall) continue; // Skip if the tile is a wall
  if (currentTile.distance === Infinity) break; // Break if the tile's distance is infinity
  currentTile.isTraversed = true; // Mark the tile as traversed
  traversedTiles.push(currentTile); // Add the tile to the traversed tiles array
  if (isEqual(currentTile, endTile)) break; // Break if the tile is the end tile
```

Initialization

- traversedTiles: Keeps track of all visited tiles.
- base: The starting tile from the grid.
- Set its distance to 0 because we're starting there.
- Mark it as visited.
- untraversedTiles: A stack used for DFS, starting with the start tile.

Check if tile is valid

- Skip if it's a wall.
- Stop if its distance is Infinity (unreachable).
- Mark it as visited and store it.
- If it's the end tile, stop the search.

DEPTH FIRST SEARCH (DFS)

Get neighbors and add them to the stack

```
const neighbors = getUntraversedNeighbors(grid, currentTile); // Get untraversed neighbors of the tile
for (let i = 0; i < neighbors.length; i += 1) {
  // Iterate through each neighbor
  if (!checkStack(neighbors[i], untraversedTiles)) {
    // Check if the neighbor is not in the stack
    neighbors[i].distance = currentTile.distance + 1; // Update the neighbor's distance
    neighbors[i].parent = currentTile; // Set the neighbor's parent to the current tile
    untraversedTiles.push(neighbors[i]); // Add the neighbor to the stack
  }
}
```

- Get the unvisited neighbor tiles.

For each neighbor:

If it's not already in the stack:

- Update its distance.
- Set its parent (so we can trace back the path).
- Push it onto the stack.

Build the final path

```
const path = []; // Initialize an array to store the path
let current = grid[endTile.row][endTile.col]; // Start from the end tile
while (current !== null) {
  // Backtrack until the start tile
  current.isPath = true; // Mark the tile as part of the path
  path.unshift(current); // Add the tile to the path
  current = current.parent!; // Move to the parent tile
}
```

- Start from the end tile and follow the parent pointers back to the start.
- Mark each tile in the path.
- Add each tile to the front of the path array (so it goes from start → end).

Time Complexity: O(V+E)

Space Complexity: O(V)

BREADTH FIRST SEARCH

This function implements the Breadth-First Search (BFS) algorithm on a grid to find the shortest path from the start tile to the end tile.

Initialization

```
export const bfs = (grid: GridType, startTile: TileType, endTile: TileType) => {
  const traversedTiles: TileType[] = []; // Initialize an array to store traversed tiles
  const base = grid[startTile.row][startTile.col]; // Get the start tile from the grid
  base.distance = 0; // Set the distance of the start tile to 0
  base.isTraversed = true; // Mark the start tile as traversed
  const unTraversed = [base]; // Initialize the queue with the start tile
```

- Create an empty array traversedTiles to keep track of the tiles that have been visited.
- Mark the start tile as traversed and set its distance to 0.
- Initialize the queue unTraversed with the start tile.

Traverse the grid using BFS (While loop)

```
while (unTraversed.length) {
  // Continue while there are untraversed tiles
  const tile = unTraversed.shift() as TileType; // Get the first tile from the queue
  if (tile.isWall) continue; // Skip if the tile is a wall
  if (tile.distance === Infinity) break; // Break if the tile's distance is infinity
  tile.isTraversed = true; // Mark the tile as traversed
  traversedTiles.push(tile); // Add the tile to the traversed tiles array
  if (isEqual(tile, endTile)) break; // Break if the tile is the end tile
```


- Create an empty array traversedTiles to keep track of the tiles that have been visited.
- Mark the start tile as traversed and set its distance to 0.
- Initialize the queue unTraversed with the start tile.

BREADTH FIRST SEARCH

This function implements the Breadth-First Search (BFS) algorithm on a grid to find the shortest path from the start tile to the end tile.

Explore untraversed neighbors

```
const neighbors = getUntraversedNeighbors(grid, tile); // Get untraversed neighbors of the tile
for (let i = 0; i < neighbors.length; i += 1) {
  // Iterate through each neighbor
  if (!isInQueue(neighbors[i], unTraversed)) {
    // Check if the neighbor is not in the queue
    const nei = neighbors[i]; // Get the neighbor tile
    nei.distance = tile.distance + 1; // Update the neighbor's distance
    nei.parent = tile; // Set the neighbor's parent to the current tile
    unTraversed.push(nei); // Add the neighbor to the queue
  }
```

For each untraversed neighbor of the current tile:

- Update the neighbor's distance (current tile distance + 1).
- Set the parent of the neighbor to the current tile (to backtrack later).
- Add the neighbor to the queue for further exploration.

Backtrack to construct the path

```
const path = []; // Initialize an array to store the path
let tile = grid[endTile.row][endTile.col]; // Start from the end tile
while (tile !== null) {
  // Backtrack until the start tile
  tile.isPath = true; // Mark the tile as part of the path
  path.unshift(tile); // Add the tile to the path
  tile = tile.parent!; // Move to the parent tile
}
```

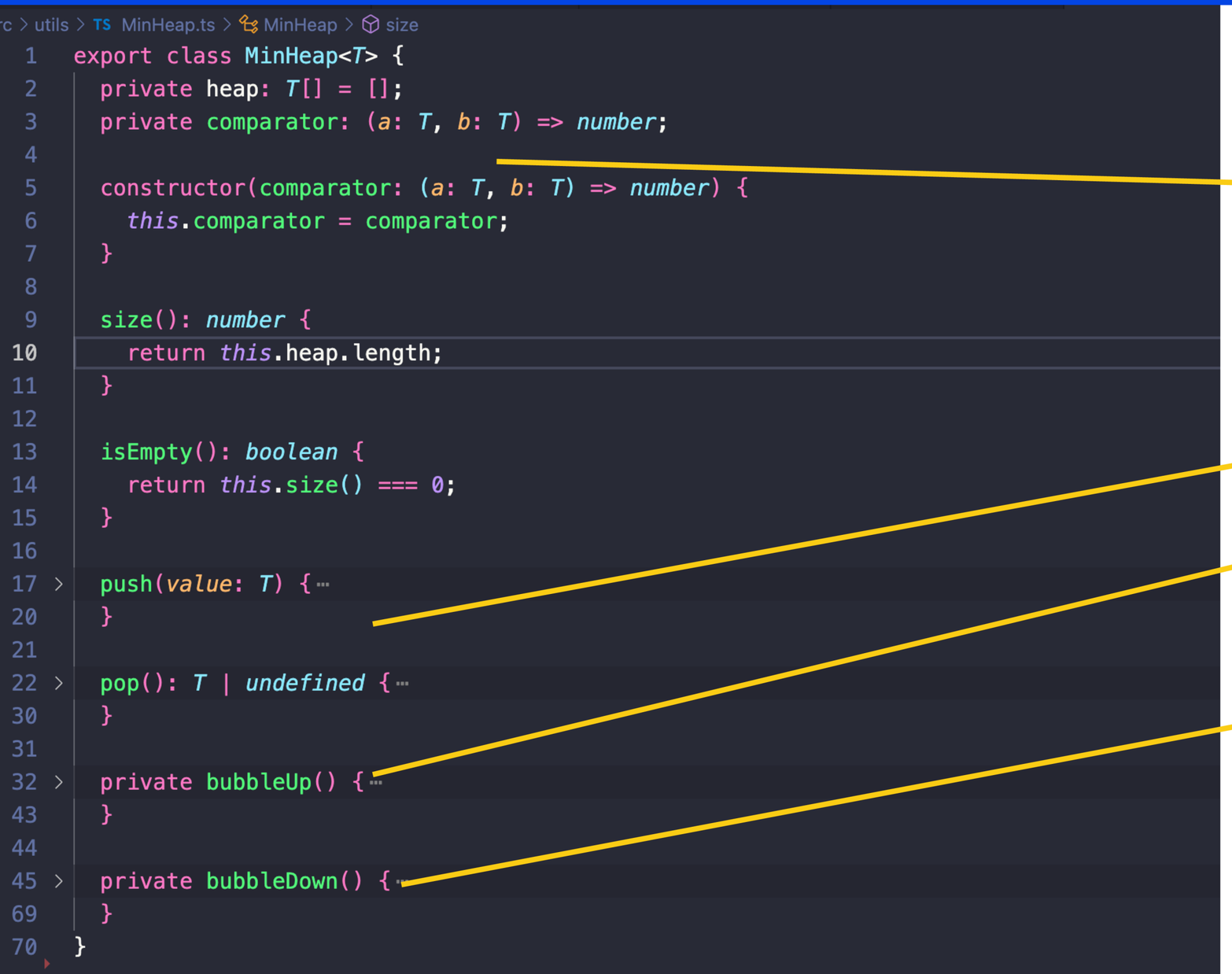
- Starting from the end tile, we backtrack through its parent references to reconstruct the path from start to end.
- Mark each tile in the path as part of the path (isPath = true).

COMPARASON

Each algorithm has its strengths depending on the problem at hand, and understanding their differences helps us choose the best one for our application.

Feature	BFS (Breadth-First Search)	DFS (Depth-First Search)
Traversal Method	Explores all nodes at the current depth level first, then moves to the next depth level (level-order).	Explores as far as possible along each branch before backtracking (depth-order).
Data Structure Used	Queue (FIFO - First In, First Out).	Stack (LIFO - Last In, First Out).
Time and Space Complexity	Time: O(V + E), Space: O(V)	Time: O(V + E), Space: O(V)

MIN-HEAP (PRIORITY QUEUE)



Min-Heap features:

- **Generic**: allow works with any type **T**, and **comparator()** function to compare two elements.
- **Heap Property:** The smallest element always at the top.
- Both **push** (insert) and **pop** (remove min) run in **O(log n)** time.
- **bubbleUp()**: Moves the last element up until the heap property is restored.
- **bubbleDown():** Moves the top element down until the heap property is restored.

DIJKSTRA'S SEARCH

1. Initialize min-heap and array to stored tile traversed

```
export const dijkstra = (
  grid: GridType,
  startTile: TileType,
  endTile: TileType
) => {
  const traversedTiles = [];
  const base = grid[startTile.row][startTile.col];
  base.distance = 0;

  // Use MinHeap for efficient extraction of the minimum distance tile
  const minHeap = new MinHeap<TileType>((a, b) => a.distance - b.distance);
  minHeap.push(base);
```

Initialization

- set startTile.distance = 0
- minHeap = new MinHeap ordered by **tile.distance**
- minHeap.push(startTile)
- traversedTiles = []

DIJKSTRA'S SEARCH

2. Main While loop

```
while (!minHeap.isEmpty()) {
  const currentTile = minHeap.pop();
  if (!currentTile) break;
  if (currentTile.isWall) continue;
  if (currentTile.isTraversed) continue; // Skip if already traversed
  if (currentTile.distance === Infinity) break;

  currentTile.isTraversed = true;
  traversedTiles.push(currentTile);

  if (isEqual(currentTile, endTile)) break;

  const neighbors = getUntraversedNeighbors(grid, currentTile);
  for (let i = 0; i < neighbors.length; i += 1) {
    if (currentTile.distance + 1 < neighbors[i].distance) {
      neighbors[i].distance = currentTile.distance + 1;
      neighbors[i].parent = currentTile;
      minHeap.push(neighbors[i]);
    }
  }
}
```

Check the tile condition until min-heap empty

- **take the smallest tile from min-heap as currentTile**
- **continue if** : it is wall or already traversed.
- **stop if:** the distance is infinity

check the untraversed neighbors of current tile:

- **if the distance to the current tile+1 is smaller neighbor: → choose the path of current tile and push the the neigbor to min-heap.**

DIJKSTRA'S SEARCH

3. Construct the shorest path from traversed tiles

```
const path = [];
let current = grid[endTile.row][endTile.col];
while (current !== null && current.parent !== undefined) {
  current.isPath = true;
  path.unshift(current);
  current = current.parent!;
}
return { traversedTiles, path };
```

- set the current tile at the target node
- come back to start point by going along with the current to their parent until the their parent is not defined

Time complexity: O(E log V)

- V = number of tiles (vertices)
- E = number of edges

Space complexity: O(V)

- For the grid, MinHeap, traversedTiles, and path arrays. Each stores up to V tiles.

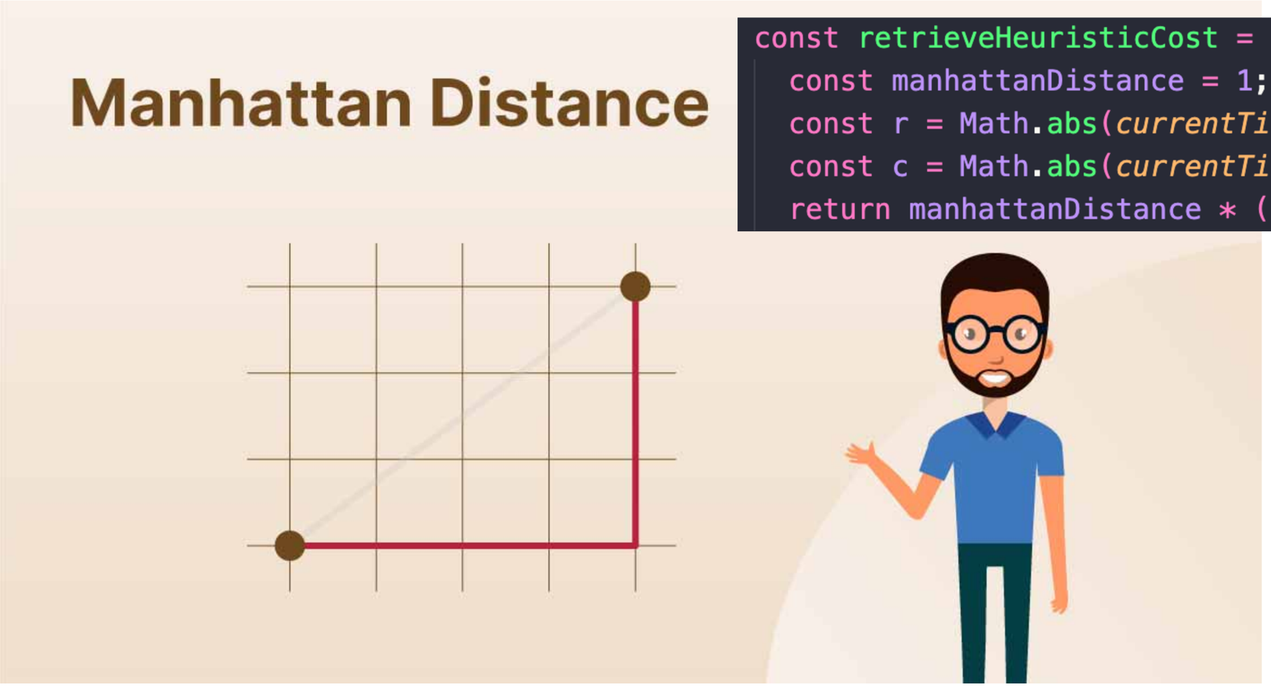
A* SEARCH

A* search overview

- **A* (A-Star)** is a best-first search algorithm that finds the shortest path from a start node to a goal node, using both:
- **g(n)**: The cost from start to current node and **h(n):** a heuristic estimate of the cost to goal
- It selects the next nodes based on: **f(n) = g(n) + h(n)**
- → This makes it complete, optimal (with an admissible heuristic), and efficient in many practical cases.

A* SEARCH

Heuristic Function




```
const retrieveHeuristicCost = (currentTile: TileType, endTile: TileType) => {
  const manhattanDistance = 1; // Define the constant multiplier for Manhattan distance
  const r = Math.abs(currentTile.row - endTile.row); // Calculate the absolute difference in rows
  const c = Math.abs(currentTile.col - endTile.col); // Calculate the absolute difference in colu
  return manhattanDistance * (r + c); // Return the Manhattan distance (sum of row and column dif
```

Manhattan Distance for current tile to the end tile

Manhattan distance is suitable for grids where movement is only allowed in four directions (up, down, left, right).

It estimates the minimum number of steps required to reach the goal, ignoring obstacles.

A* SEARCH

The heuristic array Function

```
export const initHeuristicCost = (grid: GridType, endTile: TileType) => {
  const heuristicCost = []; // Initialize an empty array to store heuristic costs
  for (let i = 0; i < MAX_ROWS; i += 1) {
    // Loop through each row in the grid
    const row = []; // Initialize an empty array to store heuristic costs for the current row
    for (let j = 0; j < MAX_COLS; j += 1) {
      // Loop through each column in the current row
      row.push(retrieveHeuristicCost(grid[i][j], endTile)); // Calculate and add the heuristic cost for the current tile
    }
    heuristicCost.push(row); // Add the row of heuristic costs to the heuristicCost array
  }
  return heuristicCost; // Return the 2D array of heuristic costs
};
```

Builds a 2D matrix that stores the Manhattan distance from each tile to endTile - **h(n)**

The total fuction cost

```
export const initFunctionCost = () => {
  const functionCost = []; // Initialize an empty array to store function costs
  for (let i = 0; i < MAX_ROWS; i += 1) {
    // Loop through each row in the grid
    const row = []; // Initialize an empty array to store function costs for the current row
    for (let j = 0; j < MAX_COLS; j += 1) {
      // Loop through each column in the current row
      row.push(Infinity); // Set the initial function cost for each tile to Infinity
    }
    functionCost.push(row); // Add the row of function costs to the functionCost array
  }
  return functionCost; // Return the 2D array of function costs
};
```

- Builds a 2D array that stores **f(n)** values for every tile.
- Initially, all function costs are set to Infinity since no path has been discovered yet.

A* SEARCH

Initialize the structure for A* search

```
export const aStar = (
  grid: GridType,
  startTile: TileType,
  endTile: TileType
) => {
  const traversedTiles: TileType[] = [];
  const heuristicCost = initHeuristicCost(grid, endTile);
  const functionCost = initFunctionCost();
  const base = grid[startTile.row][startTile.col];
  base.distance = 0;
  functionCost[base.row][base.col] = base.distance + heuristicCost[base.row][base.col];

  // Initialize the min-heap priority queue
  const openSet = new MinHeap<TileType>((a, b) => {
    const fA = functionCost[a.row][a.col];
    const fB = functionCost[b.row][b.col];
    if (fA === fB) {
      return b.distance - a.distance; // Prefer nodes with more progress if tie
    }
    return fA - fB;
  });

  openSet.push(base);
```

- store all traversedTiles into array.
- Stores precomputed **(h(n))**
- Set up starting tile - **base**
- functionCost: initially set to Infinity for all tiles.

- Uses f(n) to determine node priority
- Tie-breaker: prefer node with larger g(n) (deeper progress)

A* SEARCH

In the while loop

src > lib > algorithms > pathfinding > TS aStar.ts > aStar > openSet > <function>

```
78  export const aStar = (
102   while (!openSet.isEmpty()) {
103     const currentTile = openSet.pop();
104     if (!currentTile || currentTile.isWall) continue;
105     if (currentTile.distance === Infinity) break;
106
107     currentTile.isTraversed = true;
108     traversedTiles.push(currentTile);
109     if (isEqual(currentTile, endTile)) break;
110
111     const neighbors = getUntraversedNeighbors(grid, currentTile);
112     for (const neighbor of neighbors) {
113       const distanceToNeighbor = currentTile.distance + 1;
114       if (distanceToNeighbor < neighbor.distance) {
115         neighbor.distance = distanceToNeighbor;
116         functionCost[neighbor.row][neighbor.col] =
117           neighbor.distance + heuristicCost[neighbor.row][neighbor.col];
118         neighbor.parent = currentTile;
119         openSet.push(neighbor);
120       }
121     }
122   }
123
124   const path: TileType[] = [];
125   let current: TileType | null = grid[endTile.row][endTile.col];
126   while (current !== null) {
127     current.isPath = true;
128     path.unshift(current);
129     current = current.parent!;
130   }
```

- Take the currentTile from **min-heap.**
- Skip if it cannot be reached or end tile
- continue if it is a wall.

- Updates neighbor's g(n), f(n), and parent if a better path is found
- Adds neighbor to heap for further exploration

A* SEARCH

construct the shortest path

src > lib > algorithms > pathfinding > TS aStar.ts > aStar > openSet > <function>

```
78  export const aStar = (
102   while (!openSet.isEmpty()) {
103     const currentTile = openSet.pop();
104     if (!currentTile || currentTile.isWall) continue;
105     if (currentTile.distance === Infinity) break;
106
107     currentTile.isTraversed = true;
108     traversedTiles.push(currentTile);
109     if (isEqual(currentTile, endTile)) break;
110
111     const neighbors = getUntraversedNeighbors(grid, currentTile);
112     for (const neighbor of neighbors) {
113       const distanceToNeighbor = currentTile.distance + 1;
114       if (distanceToNeighbor < neighbor.distance) {
115         neighbor.distance = distanceToNeighbor;
116         functionCost[neighbor.row][neighbor.col] =
117           neighbor.distance + heuristicCost[neighbor.row][neighbor.col];
118         neighbor.parent = currentTile;
119         openSet.push(neighbor);
120       }
121     }
122   }
123
124   const path: TileType[] = [];
125   let current: TileType | null = grid[endTile.row][endTile.col];
126   while (current !== null) {
127     current.isPath = true;
128     path.unshift(current);
129     current = current.parent!;
130   }
```

- Starts from endTile
- Walks backward through .parent to reconstruct the shortest path

Time and SpaceComplexity

- **Time complexity: O(E log V):**
- V = number of tiles (vertices)
- E = number of edges.
- Each tile pushed/popped from the MinHeap multiple times, each operation is O(log V).
- **Space complexity: O(log V):**
- functionCost + heuristicCost + parent pointers = O(V)
- Heap space = O(V)

COMPARASON

Each algorithm has its strengths depending on the problem at hand, and understanding their differences helps us choose the best one for our application.

Feature	Dijkstra Search	A* Search
Traversal Method	Explores all possible paths equally based on actual cost, often traversing more nodes than necessary.	uses both actual cost and a heuristic to prioritize nodes, guiding the search toward the goal and reducing unnecessary exploration.
Data Structure Used	**Min-Heap (Priority Queue)** - ensures the most promising tiles are always explored first.	**Min-Heap (Priority Queue)** - ensures the most promising tiles are always explored first.
Time and Space Complexity	Time: O(E log V), Space: O(V)	Time: O(E log V), Space: O(V)

DEMO VIDEO

Fu


Fastest Path
Maze Type
Binary Tree
Algorithm Type
Breath First Search
Speed
Fast
Time: 0.00s

INTERNATIONAL UNIVERSITY
School of Computer Science and Engineering
(Sem 2, 2024 - 2025)

THANK YOU

FOR YOUR ATTENTION

Algorithms & Data Structure

Le Hung - ITCSIU22271 (Gr: 0202)

Tran Dang Nhat - ITITIU22115 (Gr: 0201)