Java Maze Solver Specification

Project Name: Optimal Maze Escape **Technology Stack:** Java Development Kit (JDK) 21 **Target Environment:** Console or Basic GUI (System.out for console, or simple Swing/JavaFX for minimal grid display) **Version:** 1.0

1. Project Goals and Scope

The primary goal of this program is to provide a complete, interactive, and analytical maze-solving application. It must fulfill two main functions:

1. **Optimal Solution:** Calculate the shortest possible path from any starting point to the designated exit.
2. **User Interaction:** Allow a user to navigate the maze interactively and compare their path length against the optimal path.

The program must be fully self-contained, handling maze generation, solution calculation, user input, and result display.

2. Technical Stack & Architecture

2.1 Technology

- **Language:** Java 21 (Leveraging modern features like Records for data structures if applicable).
- **Dependencies:** Standard JDK libraries only (no external libraries required for core functionality).

2.2 Data Structures

The application will rely on two primary data structures:

Data Structure	Purpose	Implementation Detail
Maze Grid	Stores the state of the maze (walls, open paths, start, exit).	A 2D array of integers or enums (e.g., `int[][]` or `CellType[][]`).
Queue	Used by the Breadth-First Search (BFS) algorithm to find the optimal path.	`java.util.LinkedList` or `java.util.ArrayDeque` .
Stack	Used by the Randomized Depth-First Search (DFS) algorithm for maze generation.	`java.util.Stack` or `java.util.LinkedList` .
Point/Coordinate	Represents a position within the maze (e.g., a path segment or a wall).	A Java `record` named `Point(int x, int y)` is ideal.

3. Functional Requirements

3.1 Maze Characteristics

- **Size:** Variable, square maze size (`N x N`).

- **Bounds:** `N` must be between 20 and 100 units (e.g., 20x20 up to 100x100).
- **Generation:** The maze structure itself must be randomly generated (see Algorithms).
- **Exit Point:** A single exit point (`E`) must be randomly generated on the perimeter of the maze.

3.2 User Interaction

- **Input:** Player movement is controlled via standard keyboard input.
  - `'w'` : Up
  - `'s'` : Down
  - `'a'` : Left
  - `'d'` : Right
- **Movement Logic:** The player token (`P`) must not be allowed to move into an impassable block (wall) or out of the maze boundaries. Only horizontal and vertical movement is allowed.

3.3 Display Requirements

The program must display the maze on a grid, ideally using simple characters or terminal colors if a graphical environment is not used.

- **Impassable Block (Wall):** Displayed as a **Black Square** (or `#` in console).
- **Empty Space (Path):** Displayed as a **White Square** (or `.` in console).
- **Player Position:** Displayed as a distinct token (e.g., `P`).
- **Exit Position:** Displayed as a distinct token (e.g., `E`).

3.4 Outcome and Comparison

Upon the player reaching the exit point:

1. The program must stop accepting movement input.
2. It must display the player's total step count (`C_player`).
3. It must display the optimal (fewest steps) path length (`C_optimal`).
4. It must display a comparison statement (e.g., "You took X extra steps," or "You found the optimal path!").

4. Core Algorithms

The program relies on two distinct algorithms for its core functionality:

4.1 Maze Generation (Randomized DFS Backtracking)

This algorithm will ensure the maze is traversable and contains no unreachable areas.

1. **Grid Initialization:** Initialize the grid with all cells set to **Wall**.
2. **Start Point:** Choose a random starting cell and set it to **Path**.
3. **DFS Loop:** Use a stack to track the current path.
   - While the stack is not empty:
     - Look at all unvisited neighbor cells (2 units away, skipping walls).

        - If there are unvisited neighbors, choose one randomly.
        - Remove the wall between the current cell and the chosen neighbor.
        - Mark the chosen neighbor as **Path** and push it onto the stack.
        - If there are no unvisited neighbors, backtrack (pop the current cell from the stack).

4.2 Optimal Pathfinding (Breadth-First Search - BFS)

BFS is the mandated algorithm because it guarantees finding the shortest path (in terms of number of steps) in an unweighted graph (where every move has a cost of 1).

1. **Setup:** Use a queue to store cells to visit, and a parallel 2D array (`distance[][]` or `visited[][]`) to track the distance from the exit and prevent infinite loops.
2. **Start:** Begin the search from the **Exit Point (** `E` **)**. Set the exit's distance to 0.
3. **Search:** While the queue is not empty:
    - Dequeue the current cell.
    - For each valid neighbor (Up, Left, Down, Right, and not a wall):
        - If the neighbor has not been visited, mark it as visited, set its distance (current distance + 1), and enqueue it.
4. **Result:** When the search completes, the optimal path length from any valid start position (`S`) to the exit is simply the value stored in the `distance[S.x][S.y]` array element.

5. Implementation Structure (Key Classes)

`MazeSolver` (Main Class)

- `main(String[] args)` : Handles initialization, user input loop, and final comparison display.
- `initMaze()` : Accepts user input for `N` , performs validation (20-100), and calls the generation methods.

`Maze` (Data Model)

- `int[][] grid` : The 2D array holding the maze structure.
- `Point exit` : Stores the coordinates of the exit.
- `generateMaze(int N)` : Implements the Randomized DFS Backtracking algorithm.
- `findOptimalPath(Point start)` : Implements the BFS algorithm.
- `getOptimalSteps(Point p)` : A public method to return the pre-calculated shortest path steps from a given point `p` .

`Player` (User State)

- `Point position` : Current coordinates of the player.
- `int stepCount` : Tracks the total number of moves made by the user.
- `move(char direction, Maze maze)` : Attempts to update the player's position based on input and maze walls/bounds.

`Display` (Rendering Utility)

- `render(Maze maze, Player player)` : Takes the maze and player objects and prints the grid to the console or renders the basic GUI window.