CENG 202 – Data Structures

Maze Escape Game Project Report

Course: CENG 202 – Data Structures

Students: Deniz Cem Cangöz 220401037 - Süleyman Uçar 220401079

Submitted by: Deniz Cem Cangöz

Project Title: Maze Escape Game

Project Type: Application Project

Submission Date: May 5, 2025

Table of Contents

9. Conclusion
10. References
11. Appendices

Introduction

The Maze Escape Game is a turn-based simulation that challenges multiple autonomous agents to navigate through a dynamically changing maze environment. The core problem this project addresses is the implementation of intelligent pathfinding algorithms within a complex, shifting environment with obstacles, traps, and power-ups.

This project demonstrates practical applications of various data structures and algorithms covered in the CENG 202 course, including stacks for maze generation and agent movement history, custom lists for random selections, and intelligent decision-making algorithms for autonomous agents.

The maze environment features several unique characteristics:

- Procedurally generated perfect mazes with guaranteed solutions
- Rotating corridors that periodically change the maze structure
- Traps that impede agent progress
- Power-ups that grant special abilities
- Multiple competing agents navigating simultaneously

The solution detailed in this report implements a complete game system with autonomous agent behavior, dynamic maze management, and comprehensive statistics tracking. This project serves as a practical exploration of algorithmic problem-solving in dynamic environments and illustrates how data structures can be effectively used to model complex game mechanics.

Program Interface

Running the Program

1. **Execution**:

- **Open a terminal** (Command Prompt on Windows, Terminal on macOS/Linux).
- **Navigate** to the folder where your .jar file is located:

  bash

cd /path/to/jar

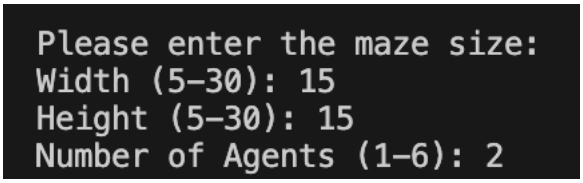
- **Run the jar**:

  bash
  java –jar MazeTile.jar

Where the parameters are:

a. width: Integer specifying the width of the maze (recommended between 10-30)
b. height: Integer specifying the height of the maze (recommended between 10-30)
c. number_of_agents: Integer specifying the number of agents (recommended between 2-6)
d. max_turns: Generated automatically based on width and height. Integer specifying the maximum number of turns

Example:



2. **Termination**: The program automatically terminates when either:
   a. All agents reach the goal
   b. The maximum number of turns is reached

The program runs in console mode and uses ASCII characters and ANSI color codes to display the maze state and agent movements.

Program Execution

Game Overview

The Maze Escape Game is a turn-based simulation where multiple AI-controlled agents navigate through a procedurally generated maze to reach a goal. Each agent takes one

turn at a time, making decisions based on the current maze state and their knowledge of the environment.

When the game begins, the following initialization steps occur:

1. A perfect maze is generated using a depth-first search algorithm.
2. Special tiles (traps, power-ups, goal) are placed throughout the maze.
3. Rotating corridors are defined in horizontal, vertical, and square patterns.
4. Agents are placed at random starting positions.

The game display shows the current state of the maze using ASCII characters and ANSI color codes:

- Walls are represented by ▉ (white)
- The goal is represented by ◉ (green)
- Empty spaces are represented by spaces
- Traps are represented by X (red)
- Power-ups are represented by their respective symbols:
    - Teleport (☒, purple)
    - Shield (⛨, blue)
    - Double Move (↝, yellow)
    - Time Extend (⏱, cyan)
- Agents are represented by their number (1, 2, 3, etc.) in their respective colors

Game Mechanics

The core game mechanics include:

1. **Turn-Based Movement**: Each agent takes one turn in rotation. During their turn, an agent can move in one of four directions (up, down, left, right) or choose to backtrack to their previous position.
2. **Pathfinding**: Agents use an AI controller to make intelligent movement decisions, prioritizing unexplored paths and avoiding traps when possible.
3. **Corridor Rotation**: Every fifth turn, a random corridor in the maze rotates, changing the maze structure and potentially altering available paths. When a corridor rotates, any agents in the corridor move with it.

4. **Trap Mechanic**: If an agent lands on a trap tile, they are forced to backtrack to their previous position unless they have a shield power-up. Traps are removed after they are triggered.
5. **Goal Achievement**: The game tracks which agents reach the goal and in what order. When an agent reaches the goal, they are removed from the turn rotation.
6. **Game Termination**: The game ends when either all agents have reached the goal or the maximum number of turns has been reached.

Power-up System

The game features four types of power-ups that agents can collect:

1. **Teleport (L)** - 25% of power-ups:
    a. Grants 2 charges of teleportation
    b. When active, has a 30% chance to teleport the agent to a random valid location
    c. Useful for escaping dead-ends or quickly traversing the maze
2. **Shield (S)** - 35% of power-ups:
    a. Provides immunity to traps for 5 turns
    b. When an agent with an active shield encounters a trap, the trap is removed without affecting the agent
3. **Double Move (D)** - 30% of power-ups:
    a. Allows the agent to move twice in the same direction for 3 turn
    b. If the second move would be invalid, only the first move is performed
4. **Time Extend (X)** - 10% of power-ups:
    a. Extends the game duration (implementation details left for future expansion)

Each power-up has a limited duration or number of uses. The game displays the active power-ups for each agent during their turn.

Corridor Rotation System

The corridor rotation system adds a dynamic element to the maze, forcing agents to adapt their pathfinding strategies:

1. **Corridor Types**:
    a. Horizontal corridors: Span the width of the maze at specific rows
    b. Vertical corridors: Span the height of the maze at specific columns
    c. Square corridors: 2x2 areas in the corners of the maze

2. **Rotation Mechanics**:
    a. Every 5 turns, a random corridor is selected to rotate
    b. The rotation shifts all tiles in the corridor, including any special tiles
    c. If an agent is on a tile that rotates, the agent moves with the tile
3. **Tactical Implications**:
    a. Rotations can open new paths or close existing ones
    b. Agents must consider the possibility of rotation when planning paths
    c. Rotations can unexpectedly move agents to new locations

The corridor rotation system creates a dynamic environment that requires agents to be adaptable and responsive to changing maze conditions.

Input and Output

Input Format

The Maze Escape Game accepts command-line arguments for configuration:

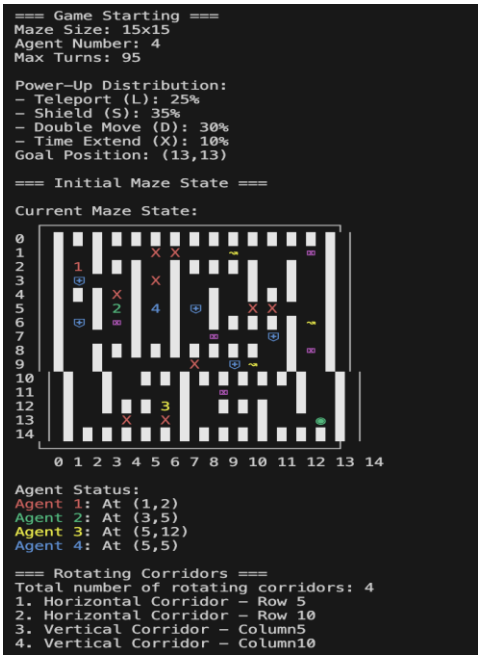
java Main [width] [height] [number_of_agents] [max_turns]

- width: Integer between 10-30
- height: Integer between 10-30
- number_of_agents: Integer between 2-6
- max_turns: Generated automatically: Integer 50 + (width * height / 5);

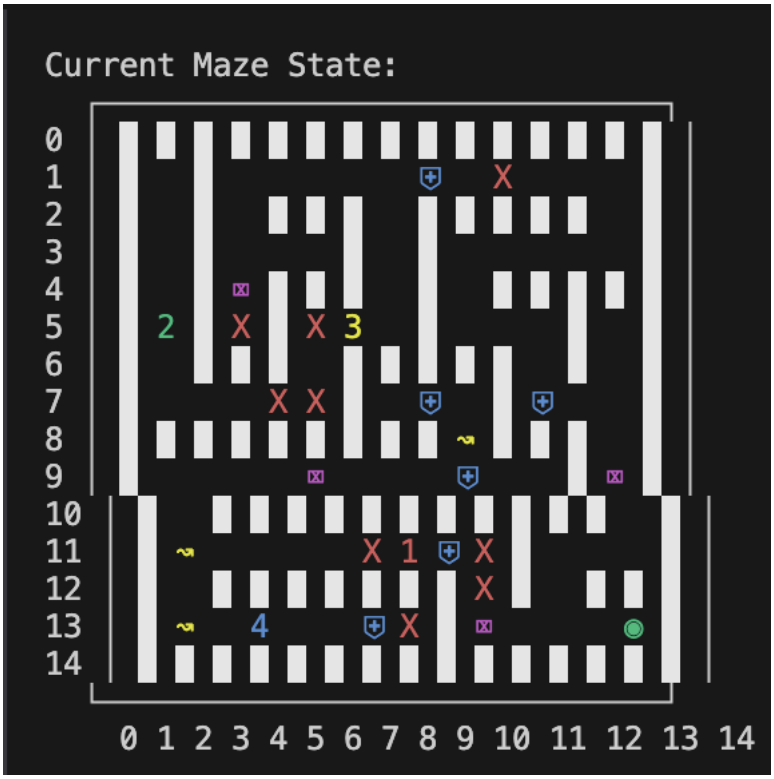
Output Format

The game output is presented in the console with ANSI color formatting. The output includes:

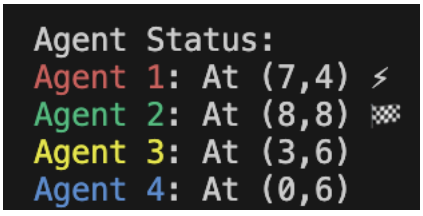
1. **Initial Configuration Display**:



2. **Maze Display**:



3. **Agent Status**:



4. **Turn Information**:

=== Turn 25: CORRIDOR ROTATION ===
Corridor rotated: vertical at column 6
Agent 1 restored to original position
Rotated vertical corridor at column 6
Agent 1 moved RIGHT
Agent 1 found a power-up!
Agent 1 got Shield (5 turns)!

5. **Final Results**:

```
=== Game Over ===
Total turns: 70

Goal Achievement Order:
1. Player 2 (Turn 8)
2. Player 4 (Turn 9)
3. Player 1 (Turn 14)

Detailed Statistics:
+------+-------+-------+--------+------------+----------------+
| Player | Moves | Traps | PwrUps | Backtracks | Goal Turn |
+------+-------+-------+--------+------------+----------------+
| 1      | 14    | 0     | 0      | 0          | 14             |
| 2      | 8     | 1     | 0      | 1          | 8              |
| 3      | 10    | 0     | 1      | 10         | DNF            |
| 4      | 9     | 0     | 0      | 0          | 9              |
+------+-------+-------+--------+------------+----------------+
Game summary successfully written to game_summary.txt
```

Output Files

In addition to console output, the game generates a text file named game_summary.txt at the end of execution. This file contains a summary of the game, including:

1. Game configuration details
2. Game outcome (total turns played)
3. Goal achievement order
4. Detailed statistics for each agent

Example file content:

game_summary.txt

```
=== Maze Escape Game Summary ===
Generated on: 2025-05-04 17:06:01

Game Configuration:
- Maze Size: 10x10
- Number of Agents: 4
- Max Turns: 70
- Goal Position: (8,8)

Game Outcome:
- Total Turns Played: 70

Goal Achievement Order:
1. Player 2 (Turn 8)
2. Player 4 (Turn 9)
3. Player 1 (Turn 14)

Detailed Statistics:
+--------+-------+-------+--------+------------+-------------+
| Player | Moves | Traps | PwrUps | Backtracks | Goal Reached |
+--------+-------+-------+--------+------------+-------------+
| 1      | 14    | 0     | 0      | 0          | Yes         |
| 2      | 8     | 1     | 0      | 1          | Yes         |
| 3      | 10    | 0     | 1      | 10         | No          |
| 4      | 9     | 0     | 0      | 0          | Yes         |
+--------+-------+-------+--------+------------+-------------+


=== End of Summary ===

```

Program Structure

Class Diagram

The Maze Escape Game consists of several key classes with the following relationships:

UML Diagram Link: https://imgur.com/a/XzQVPix

Class Descriptions

1. **App**

The main entry point for the Maze Escape Game application. It handles user interface for configuring game parameters like maze dimensions and number of agents. The class collects valid user input through console interaction and initializes the game by creating a GameController instance.

2. **GameController**

The central controller for the maze game that manages game flow and coordinates between various components. It maintains references to the maze, agents, and handles turn progression. This class tracks game state including turn count and maximum allowed turns, processes agent movements, and handles special cases like teleportation. It's responsible for starting the autonomous game mode and logging game results.

3. **MazeManager**

Responsible for creating and managing the maze structure including the grid of tiles, agent positions, and special features like rotating corridors. It generates the perfect maze, places special tiles (traps, power-ups, goal), and handles the logic for valid movements. This class also manages the dynamic aspects of the maze such as corridor rotation and checking tile effects when agents land on them.

4. **MazeTile**

Represents an individual cell within the maze grid. Each tile has a specific type (empty, wall, trap, goal, or various power-ups) and tracks whether an agent occupies it. The class provides methods to query and modify tile state, including determining if it's traversable and handling agent presence.

5. **Agent**

Models a player or AI entity navigating the maze. It tracks position, movement history, collected power-ups, and goal achievement status. The class maintains state for various power-up effects like teleport charges, shield protection, and double movement ability, and provides methods to apply and manage these effects.

6. **CircularCorridor**

Represents a section of the maze that can rotate, adding dynamic complexity to gameplay. It manages a collection of tiles that form either horizontal, vertical, or square corridors. The class provides rotation functionality that changes the maze configuration during gameplay, requiring agents to adapt their pathing strategies.

7. **AIController**

Implements the artificial intelligence for autonomous agents. This class uses pathfinding and decision-making algorithms to determine optimal movement for agents navigating the maze. It references the maze state and controls agent movement decisions to reach the goal efficiently while avoiding traps.

8. **PlayerRegistry**

A data structure that registers and maintains references to all agents in the game. It provides access to agents by index, supporting the game controller's management of multiple agents.

9. **TurnManager**

Handles the turn-based progression of the game. It maintains a queue of agents and determines which agent can act next. The class tracks game rounds, rotates through agents, checks for game completion conditions, and logs summaries of each turn's actions.

10. **CustomList**

A generic list implementation used throughout the system for storing and managing collections of objects. It provides basic list operations like adding elements, retrieving by index, and checking if the list is empty.

11. **CustomStack**

A specialized stack data structure primarily used for tracking agent movement history to support features like backtracking. It implements standard stack operations (push, pop, peek) using a linked node structure.

12. **CustomQueue**

A queue implementation used by the TurnManager to maintain the order of agent turns. It supports standard queue operations (enqueue, dequeue, peek) and provides a method to check if all agents in the queue have reached their goal.

13. **Stack**

A specialized stack implementation for storing coordinates or other integer array data. It's used in maze generation and pathfinding algorithms.

Data Structures

The project implements several key data structures:

1. CustomList

Implementation Details:

- Based on a singly-linked list using nodes
- Each node contains a data element and a reference to the next node
- Maintains a head reference and size counter

Operations:

- add(Object data): Appends an element to the end of the list in O(n) time
- get(int index): Retrieves an element at a specific position in O(n) time
- size(): Returns the number of elements in O(1) time
- isEmpty(): Checks if the list contains no elements in O(1) time
- removeRandom(): Removes and returns a random element from the list in O(n) time

Usage in Project:

- Used by MazeManager to maintain the list of rotating corridors
- Provides flexibility for dynamic addition of special maze features
- The removeRandom() method enables random corridor selection for rotation events

2. CustomStack

Implementation Details:

- Implemented as a singly-linked list with LIFO (Last-In-First-Out) behavior
- Each node contains a string data element and reference to the node below it
- Maintains a top reference and size counter

Operations:

- push(String data): Adds an element to the top of the stack in O(1) time
- pop(): Removes and returns the top element in O(1) time

- peek(): Returns the top element without removing it in O(1) time
- isEmpty(): Checks if stack contains no elements in O(1) time
- size(): Returns the number of elements in O(1) time
- getTop(): Returns the top node reference

Usage in Project:

- Used by Agent to record movement history as a sequence of directions
- Enables the backtracking feature by allowing agents to retrace their steps
- Movement sequences can be popped off to reverse agent movement when necessary

3. CustomQueue

Implementation Details:

- Implemented as a linked structure with FIFO (First-In-First-Out) behavior
- Maintains front and rear node references for efficient operations
- Tracks size for quick element count access

Operations:

- enqueue(Agent agent): Adds an agent to the back of the queue in O(1) time
- dequeue(): Removes and returns the front agent in O(1) time
- peek(): Returns the front agent without removing it in O(1) time
- isEmpty(): Checks if queue contains no agents in O(1) time
- size(): Returns the number of agents in O(1) time
- allAgentsFinished(): Checks if all agents in the queue have reached the goal

Usage in Project:

- Used by TurnManager to manage the order of agent turns
- Allows for fair distribution of turns among multiple agents
- Supports the round-robin scheduling of agent movements
- Helps track whether all agents have completed the maze

4. Stack (Integer Array Stack)

Implementation Details:

- Specialized stack implementation for storing integer arrays (typically coordinates)

- Uses a linked structure where each node contains an int[] and a reference to the next node
- Maintains a top reference and size counter

Operations:

- push(int[] data): Adds coordinates to the top of the stack in O(1) time
- pop(): Removes and returns the top coordinates in O(1) time
- peek(): Returns the top coordinates without removing it in O(1) time
- isEmpty(): Checks if stack contains no elements in O(1) time

Usage in Project:

- Used in maze generation algorithms (likely for depth-first search)
- Tracks cell coordinates during the perfect maze generation process
- Stores position information when implementing pathfinding algorithms
- Used to track positions during corridor rotation to update agent locations after rotation

5. PlayerRegistry

Implementation Details:

- Custom registry structure based on a linked list
- Nodes contain references to Agent objects
- Maintains a head reference and size counter

Operations:

- register(Agent agent): Adds an agent to the registry in O(1) time
- getAgent(int index): Retrieves an agent by index in O(n) time
- Implicitly tracks the number of registered agents

Usage in Project:

- Used by GameController to maintain a registry of all agents in the game
- Provides access to agents by index for turn management
- Supports the game's ability to work with a variable number of agents

Node Class

The Node class appears to be a fundamental building block used across multiple data structures:

- Acts as a container for data elements in the linked structures
- Contains a data field (varies based on structure) and a next reference
- Enables the creation of chains of references to form lists, stacks, and queues

Algorithms

1. **Maze Generation Algorithm**:
    a. Uses depth-first search with backtracking
    b. Ensures a perfect maze (exactly one path between any two points)
    c. Produces a maze with no isolated sections
2. **Agent Decision Making**:
    a. Employs a priority-based decision algorithm
    b. Avoids previously visited paths when possible
    c. Weighs decisions based on goal proximity and trap avoidance
3. **Corridor Rotation Algorithm**:
    a. Selects corridors based on predefined patterns
    b. Rotates tiles while maintaining their properties
    c. Updates agent positions when affected by rotation
4. **Pathfinding**:
    a. Uses a simplified heuristic approach
    b. Considers both exploration and goal-seeking behavior
    c. Adapts to changing maze conditions

Examples

Game Execution Example

The following is an abbreviated example of game execution with 4 agents on a 15x15 maze:

1. **Game Initialization**:

=== Game Starting ===
Maze Size: 15x15
Agent Number: 4
Max Turns: 100

Power-Up Distribution:
- Teleport (L): 25%

- Shield (S): 35%
- Double Move (D): 30%
- Time Extend (X): 10%
Goal Position: (13,13)

=== Initial Maze State ===
[Maze display with initial positions]

Initialized 8 rotating corridors

2. **First Few Turns**:

=== Turn 1 ===
Agent 1 moved RIGHT
[Updated maze display]
Player 1 at (2,1) | Moves: 1 | Backtracks: 0
Power-ups: None
Status: Still exploring

=== Turn 2 ===
Agent 2 moved UP
[Updated maze display]
Player 2 at (6,2) | Moves: 1 | Backtracks: 0
Power-ups: None
Status: Still exploring

3. **Encounter with Power-up**:

=== Turn 3 ===
Agent 3 moved LEFT
Agent 3 found a power-up!
Agent 3 got Shield (5 turns)!
[Updated maze display]
Player 3 at (7,10) | Moves: 1 | Backtracks: 0
Power-ups: Shield
Status: Still exploring

4. **Corridor Rotation**:

=== Turn 5: CORRIDOR ROTATION ===
Rotated horizontal corridor at row 5
[Updated maze display showing rotated corridor]

5. **Trap Encounter**:

=== Turn 7 ===
Agent 1 moved DOWN
Agent 1 hit a trap!
Agent 1 backtracked to (3,2)!
[Updated maze display]
Player 1 at (3,2) | Moves: 3 | Backtracks: 1
Power-ups: None
Status: Still exploring

6. **Goal Achievement**:

=== Turn 23 ===
Agent 4 moved RIGHT
Agent 4 reached the goal!
[Updated maze display]
Player 4 at (13,13) | Moves: 23 | Backtracks: 0
Power-ups: None
Status: REACHED GOAL! 🏁

7. **Game Conclusion**:

=== Game Over ===
Total turns: 85

Goal Achievement Order:
1. Player 4 (Turn 23)
2. Player 2 (Turn 41)
3. Player 1 (Turn 63)
4. Player 3 (Turn 78)
[Detailed statistics table]
Game summary successfully written to game_summary.txt

Agent Decision Making

The following example demonstrates the decision-making process of an agent in a specific maze scenario:

Consider an agent at position (5,7) with the following surroundings:

- Wall to the north
- Empty space to the east
- Trap to the south
- Previously visited path to the west
- Goal at position (13,13)

The AI controller analyzes the situation:

1. First, it identifies available moves:
    a. North: Invalid (wall)
    b. East: Valid (empty space)
    c. South: Valid but risky (trap)
    d. West: Valid but previously visited
2. It calculates distances to the goal:
    a. Current distance: $\sqrt{((5-13)^2 + (7-13)^2)} \approx 10.8$
    b. If move east: $\sqrt{((6-13)^2 + (7-13)^2)} \approx 10.0$ (closer)
    c. If move south: $\sqrt{((5-13)^2 + (8-13)^2)} \approx 10.3$ (closer)
    d. If move west: $\sqrt{((4-13)^2 + (7-13)^2)} \approx 11.7$ (farther)
3. It applies priority rules:
    a. Avoids traps when possible
    b. Prioritizes unvisited paths
    c. Prefers moves that reduce distance to goal
4. Decision:
    a. East is chosen as the optimal move (unvisited, no trap, reduces distance to goal)

This example illustrates how agents evaluate their surroundings and make strategic decisions to navigate the maze efficiently.

Improvements and Extensions

While the Maze Escape Game successfully implements the core functionality, several potential improvements and extensions could enhance the system:

1. **User Interface Enhancements**:

    a. A graphical user interface could provide a more intuitive visualization of the maze
    b. Animation of agent movements and corridor rotations would improve visual feedback
    c. Interactive controls to pause, resume, or adjust game speed would enhance user experience
2. **Advanced AI Strategies**:
    a. Implement multiple AI strategies (e.g., A* pathfinding, wall-following, random)
    b. Add learning capabilities to agents that improve over multiple games
    c. Introduce cooperation or competition mechanics between agents
3. **Additional Game Mechanics**:
    a. Implement "doors" and "keys" requiring agents to collect keys to open paths
    b. Add moving obstacles that patrol certain areas of the maze
    c. Introduce agent-specific abilities or characteristics
4. **Customization Options**:
    a. Allow users to design custom mazes rather than only procedurally generated ones
    b. Provide options to adjust power-up distribution and frequency
    c. Enable customization of trap effects and corridor rotation patterns
5. **Multiplayer Support**:
    a. Add support for human players to compete against AI agents
    b. Implement network play for multiple human players
    c. Create a scoring system and leaderboard
6. **Expanded Statistics and Analysis**:
    a. Generate heatmaps showing agent movement patterns
    b. Track and visualize decision-making effectiveness
    c. Provide more detailed analysis of agent performance
7. **Performance Optimization**:
    a. Optimize the maze generation algorithm for larger maze sizes
    b. Improve rendering performance for faster visual updates
    c. Enhance memory management for long-running games

Some of these features were initially planned but could not be implemented due to time constraints or scope limitations. These improvements represent natural evolution points for future versions of the game.

Difficulties Encountered

During the development of the Maze Escape Game, we faced several challenges that impacted the implementation process:

1. **Maze Generation Algorithm**:
    a. Implementing a perfect maze generation algorithm proved more complex than anticipated
    b. Ensuring connectivity between all maze sections required careful handling of edge cases
    c. Balancing maze complexity with solvability was a challenging optimization problem
2. **Corridor Rotation Mechanics**:
    a. The corridor rotation system introduced unexpected edge cases, particularly when agents were positioned on rotating tiles
    b. Maintaining consistent game state during rotations required careful synchronization of multiple data structures
    c. Visualizing rotations clearly in ASCII format presented formatting challenges
3. **Agent Decision-Making Logic**:
    a. Creating intelligent but not overly complex AI for agents required multiple iterations
    b. Balancing exploration with goal-seeking behavior proved difficult to tune optimally
    c. Preventing agents from getting stuck in cyclical patterns required special handling
4. **Trap and Power-up Interactions**:
    a. Managing multiple simultaneous power-up effects introduced state tracking complexity
    b. Ensuring traps functioned correctly with shield power-ups required careful testing
    c. The teleport power-up occasionally led to unintended outcomes due to randomized destination selection
5. **Data Structure Limitations**:
    a. The custom stack implementation needed careful handling to prevent memory leaks

    b. Managing agent movement history efficiently required balancing between memory usage and functionality
    c. The 2D array representation of the maze limited some dynamic operations
6. **Visualization Challenges**:
    a. ANSI color codes improved visualization but caused compatibility issues in some terminal environments
    b. Representing multiple special tiles and agents in a clear manner required iterative design
    c. Maintaining consistent formatting across different maze sizes posed layout challenges

These difficulties led to several design changes throughout the development process. For instance, we simplified some aspects of the corridor rotation system and adjusted the agent decision-making algorithm to be more deterministic. Despite these challenges, working through these problems provided valuable insights into data structure selection, algorithm design, and game mechanics implementation.

Conclusion

The Maze Escape Game project successfully demonstrates the practical application of various data structures and algorithms in creating a dynamic, interactive simulation. Through the implementation of this project, we have gained deeper understanding of several key concepts:

1. **Algorithmic Problem-Solving**: The project required implementing and adapting algorithms for maze generation, agent pathfinding, and dynamic environment modifications. This reinforced our understanding of how algorithms can be applied to solve complex problems.
2. **Data Structure Selection**: Choosing appropriate data structures for different aspects of the game (2D arrays for the maze, stacks for movement history, lists for corridor management) highlighted the importance of selecting the right tool for each task.
3. **State Management**: Tracking and managing the state of multiple agents, tiles, and power-ups in a changing environment required careful design of state management systems and demonstrated the importance of encapsulation and organization in code.
4. **Dynamic Environments**: The corridor rotation system showcased how to implement and manage dynamic environments that change during execution, presenting unique challenges for both implementation and agent behavior.

5. **AI Decision-Making**: Developing the agent decision-making logic provided practical experience in creating intelligent behavior within constrained environments and demonstrated different approaches to pathfinding and exploration.

The project successfully meets its core requirements, creating a functional and engaging maze navigation game with autonomous agents, special tiles, and dynamic maze modifications. The implementation balances complexity with performance, creating a system that is both interesting to observe and technically sound.

This project serves as a solid foundation for further exploration of more advanced game development concepts, AI behavior, and dynamic environment simulation. The modular design allows for future extensions and improvements while maintaining the core functionality established in this version.

Through the challenges encountered and solutions developed, this project has reinforced the importance of algorithmic thinking, careful data structure selection, and iterative design in software development.

References

1. ***https://youtu.be/Y37-gB83HKE?si=S7B8J5RlXWRNpSa0***

Appendices

The complete source code for the Maze Escape Game:

1)App Class:

```
import java.util.Scanner;
public class App {
public static void main(String[] args) throws Exception {
Scanner scanner = new Scanner(System.in);
System.out.println("=== Maze Escape Game ===");
System.out.println("Power-Up Info:");
System.out.println("- L (Purple ☒): Teleport - Teleporting to a random tile");
System.out.println("- S (Blue ⛨): Shield - Immune to traps for 5 rounds");
System.out.println("- D (Yellow ⇝): Double Move - 2 times faster movement for 3 rounds");
System.out.println("- X (Cyan ⏱): Time Extend - Increases the game rounds for player");
System.out.println("\nPlease enter the maze size:");
int width = getValidInput(scanner, "Width (5-30): ", 5, 30);
int height = getValidInput(scanner, "Height (5-30): ", 5, 30);
int numAgents = getValidInput(scanner, "Number of Agents (1-6): ", 1, 6);
int maxTurns = 50 + (width * height / 5); // Labirent boyutuna göre otomatik ayar
GameController game = new GameController(width, height, numAgents, maxTurns);
game.startAutonomousGame();
scanner.close();
```

```
}
private static int getValidInput(Scanner scanner, String prompt, int min, int max) {
int value;
do {
System.out.print(prompt);
while (!scanner.hasNextInt()) {
System.out.println("Invalid entry! Please enter a number.");
scanner.next();
}
value = scanner.nextInt();
} while (value < min || value > max);
return value;
}
}
```

2)Agent Class:

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
public class Agent {
private final int id;
private int currentX, currentY;
private boolean hasReachedGoal;
private boolean hasPowerUp;
private int totalMoves;
private int backtracks;
private CustomStack moveHistory;
private int trapsTriggered;
private int powerUpsCollected;
private int goalReachedTurn = -1;
public void incrementTrapsTriggered() {
trapsTriggered++;
}
public void incrementPowerUpsCollected() {
powerUpsCollected++;
}
public void setGoalReachedTurn(int turn) {
goalReachedTurn = turn;
}
public int getGoalReachedTurn() {
return goalReachedTurn;
}
public int getTrapsTriggered() {
return trapsTriggered;
}
public int getPowerUpsCollected() {
return powerUpsCollected;
}
public Agent(int id, int startX, int startY) {
this.id = id;
this.currentX = startX;
this.currentY = startY;
this.hasReachedGoal = false;
this.hasPowerUp = false;
this.totalMoves = 0;
```

```
this.backtracks = 0;
this.moveHistory = new CustomStack();
recordMove(startX, startY);
// Initialize power-up fields
this.teleportCharges = 0;
this.shieldTurnsRemaining = 0;
this.doubleMoveTurnsRemaining = 0;
this.isShieldActive = false;
}
private void recordMove(int x, int y) {
moveHistory.push(x + "," + y);
}
public void move(String direction) {
recordMove(currentX, currentY);
totalMoves++;
}
public void backtrack() {
if (moveHistory.size() > 1) {
// Remove current position
moveHistory.pop();
// Get previous position
String[] coords = moveHistory.peek().split(",");
this.currentX = Integer.parseInt(coords[0]);
this.currentY = Integer.parseInt(coords[1]);
backtracks++;
}
}
public List<String> getMoveHistoryAsList() {
List<String> history = new ArrayList<>();
CustomStack.Node current = moveHistory.getTop();
while (current != null) {
history.add(current.data);
current = current.next;
}
Collections.reverse(history);
return history;
}
public int getId() { return id; }
public int getCurrentX() { return currentX; }
public int getCurrentY() { return currentY; }
public void setCurrentX(int x) { currentX = x; }
public void setCurrentY(int y) { currentY = y; }
public boolean hasReachedGoal() { return hasReachedGoal; }
public boolean hasPowerUp() { return hasPowerUp; }
public void setPowerUp(boolean powerUp) { hasPowerUp = powerUp; }
public int getTotalMoves() { return totalMoves; }
public int getBacktracks() { return backtracks; }
private int teleportCharges;
private int shieldTurnsRemaining;
private int doubleMoveTurnsRemaining;
private boolean isShieldActive;
public void applyPowerUp(char powerUpType) {
this.hasPowerUp = true;
switch(powerUpType) {
case MazeTile.POWERUP_TELEPORT:
teleportCharges += 2;
System.out.println("Agent " + (id+1) + " got Teleport (2 charges)!");
break;
```

```
case MazeTile.POWERUP_SHIELD:
isShieldActive = true;
shieldTurnsRemaining = 5;
System.out.println("Agent " + (id+1) + " got Shield (5 turns)!");
break;
case MazeTile.POWERUP_DOUBLE_MOVE:
doubleMoveTurnsRemaining = 3;
System.out.println("Agent " + (id+1) + " got Double Move (3 turns)!");
break;
case MazeTile.POWERUP_TIME_EXTEND:
// Add game logic for time extension
System.out.println("Agent " + (id+1) + " got Time Extend!");
break;
}
}
public void setReachedGoal(boolean reached) {
this.hasReachedGoal = reached;
if (reached) {
System.out.println("Agent " + (id+1) + " has reached to target and waiting for its turn!");
}
}
public boolean canDoubleMove() {
return doubleMoveTurnsRemaining > 0;
}
public boolean hasShield() {
return isShieldActive;
}
public boolean canTeleport() {
return teleportCharges > 0;
}
public void useTeleport() {
if (teleportCharges > 0) {
teleportCharges--;
}
}
public void decrementPowerUpEffects() {
if (isShieldActive && --shieldTurnsRemaining <= 0) {
isShieldActive = false;
System.out.println("Agent " + (id+1) + "'s shield expired!");
}
if (doubleMoveTurnsRemaining > 0) {
doubleMoveTurnsRemaining--;
if (doubleMoveTurnsRemaining == 0) {
System.out.println("Agent " + (id+1) + "'s double move expired!");
}
}
hasPowerUp = teleportCharges > 0 || isShieldActive || doubleMoveTurnsRemaining > 0;
}
}
```

3)AI Conroller Class

```
import java.util.*;
import java.util.ArrayList;
import java.util.List;
public class AIController {
private final MazeManager maze;
private final Agent agent;
```

```
public AIController(MazeManager maze, Agent agent) {
this.maze = maze;
this.agent = agent;
}
public String decideSmartMove() {
int startX = agent.getCurrentX();
int startY = agent.getCurrentY();
int goalX = maze.getGoalX();
int goalY = maze.getGoalY();
Queue<Node> queue = new LinkedList<>();
boolean[][] visited = new boolean[maze.getWidth()][maze.getHeight()];
Node[][] parents = new Node[maze.getWidth()][maze.getHeight()];
queue.add(new Node(startX, startY));
visited[startX][startY] = true;
int[][] directions = {
{0, -1}, // UP
{1, 0}, // RIGHT
{0, 1}, // DOWN
{-1, 0} // LEFT
};
while (!queue.isEmpty()) {
Node current = queue.poll();
if (current.x == goalX && current.y == goalY) {
break;
}
for (int[] dir : directions) {
int nx = current.x + dir[0];
int ny = current.y + dir[1];
if (nx >= 0 && nx < maze.getWidth() && ny >= 0 && ny < maze.getHeight()) {
if (!visited[nx][ny] && maze.isValidMove(current.x, current.y, getDirection(dir))) {
visited[nx][ny] = true;
queue.add(new Node(nx, ny));
parents[nx][ny] = current;
}}}}
List<Node> path = new ArrayList<>();
Node step = new Node(goalX, goalY);
while (step != null && !(step.x == startX && step.y == startY)) {
path.add(step);
step = parents[step.x][step.y];
}
if (path.isEmpty()) {
return "BACKTRACK"; // Ulaşamıyorsa
}
Node nextStep = path.get(path.size() - 1);
return getDirectionToMove(startX, startY, nextStep.x, nextStep.y);
}
private static class Node {
int x, y;
Node(int x, int y) {
this.x = x;
this.y = y;
}
}
private String getDirection(int[] dir) {
if (Arrays.equals(dir, new int[]{0, -1})) return "UP";
if (Arrays.equals(dir, new int[]{1, 0})) return "RIGHT";
if (Arrays.equals(dir, new int[]{0, 1})) return "DOWN";
if (Arrays.equals(dir, new int[]{-1, 0})) return "LEFT";
```

```
return "";
}
private String getDirectionToMove(int fromX, int fromY, int toX, int toY) {
if (toX == fromX && toY == fromY - 1) return "UP";
if (toX == fromX && toY == fromY + 1) return "DOWN";
if (toX == fromX + 1 && toY == fromY) return "RIGHT";
if (toX == fromX - 1 && toY == fromY) return "LEFT";
return "BACKTRACK"; }
```

4)Circular Corridor Class:

```
public class CircularCorridor {
private Node head;
private int size;
private int row; // -1 means vertical corridor
private int col; // -1 means horizontal corridor
private int x; // For square blocks
private int y; // For square blocks
private CorridorType type;
private static class Node {
MazeTile data;
Node next;
Node(MazeTile data) { this.data = data; }
}
public enum CorridorType {
HORIZONTAL, VERTICAL, SQUARE
}
public CircularCorridor(MazeTile[] tiles, int rowOrX, int colOrY, CorridorType type) {
if (tiles == null || tiles.length == 0) throw new IllegalArgumentException("Tiles array cannot be empty");
this.type = type;
if (type == CorridorType.HORIZONTAL) {
this.row = rowOrX;
this.col = -1;
} else if (type == CorridorType.VERTICAL) {
this.col = colOrY;
this.row = -1;
} else {
this.x = rowOrX;
this.y = colOrY;
}
this.head = new Node(tiles[0]);
Node current = head;
for (int i = 1; i < tiles.length; i++) {
current.next = new Node(tiles[i]);
current = current.next;
}
current.next = head;
this.size = tiles.length;
}
public void rotate() {
if (head == null || head.next == head) return; MazeTile firstTile = head.data;
Node current = head;
while (current.next != head) {
current.data = current.next.data;
current = current.next;
}
```

```
current.data = firstTile;
System.out.println("Corridor rotated: " +
(type == CorridorType.HORIZONTAL ? "horizontal at row " + row :
type == CorridorType.VERTICAL ? "vertical at column " + col :
"square at (" + x + "," + y + ")"));
}
public MazeTile[] getTiles() {
MazeTile[] result = new MazeTile[size];
Node current = head;
for (int i = 0; i < size; i++) {
result[i] = current.data;
current = current.next;
}
return result;}
public boolean isHorizontal() {
return type == CorridorType.HORIZONTAL;
}
public boolean isVertical() {
return type == CorridorType.VERTICAL;
}
public boolean isSquare() {
return type == CorridorType.SQUARE;
}
public int getRow() {
if (!isHorizontal()) throw new IllegalStateException("Not a horizontal corridor");
return row;
}
public int getCol() {
if (!isVertical()) throw new IllegalStateException("Not a vertical corridor");
return col;
}
public int getX() {
if (!isSquare()) throw new IllegalStateException("Not a square corridor");
return x;
}
public int getY() {
if (!isSquare()) throw new IllegalStateException("Not a square corridor");
return y;
}}
```

5)Custom List Class:

```
class CustomList {
private int[][] items;
private int size;
private int capacity;
public CustomList() {
capacity = 10;
items = new int[capacity][4]; // Each item has 4 coordinates (x, y, wallX, wallY)
size = 0;
}
public void add(int[] item) {
if (size == capacity) {
// Resize the array if needed
capacity *= 2;
int[][] newItems = new int[capacity][4];
```

```
for (int i = 0; i < size; i++) {
System.arraycopy(items[i], 0, newItems[i], 0, 4);
}
items = newItems;
}
System.arraycopy(item, 0, items[size], 0, 4);
size++;
}
public int[] removeRandom() {
if (isEmpty()) throw new IllegalStateException("List is empty");
int randomIndex = (int)(Math.random() * size);
int[] result = new int[4];
System.arraycopy(items[randomIndex], 0, result, 0, 4);
if (randomIndex < size - 1) {
System.arraycopy(items[size - 1], 0, items[randomIndex], 0, 4);
}
size--;
return result;
}
public boolean isEmpty() {
return size == 0;
}
public int size() {
return size;
}}
```

6) Custom Queue Class:

```
class CustomQueue {
private Node front, rear;
private int size;
private static class Node {
Agent data;
Node next;
Node(Agent data) {
this.data = data;
this.next = null;
}
}
public CustomQueue() {
front = rear = null;
size = 0;
}
public void enqueue(Agent item) {
Node newNode = new Node(item);
if (isEmpty()) {
front = rear = newNode;
} else {
rear.next = newNode;
rear = newNode;
}
size++;
}
public Agent dequeue() {
if (isEmpty()) throw new IllegalStateException("Queue is empty");
Agent item = front.data;
front = front.next;
if (front == null) rear = null;
```

```
size--;
return item;
}
public Agent peek() {
if (isEmpty()) throw new IllegalStateException("Queue is empty");
return front.data;
}
public boolean isEmpty() {
return front == null;
}
public int size() {
return size;
}
public boolean allAgentsFinished() {
Node current = front;
while (current != null) {
if (!current.data.hasReachedGoal()) {
return false;
}
current = current.next;
}
return true;
}}
```

7) Custom Stack Class:

```
class CustomStack {
public static class Node {
String data;
Node next;
Node(String data) {
this.data = data;
this.next = null;
}
}
private Node top;
private int size;
public CustomStack() {
top = null;
size = 0;
}
public void push(String item) {
Node newNode = new Node(item);
newNode.next = top;
top = newNode;
size++;
}
public String pop() {
if (isEmpty()) throw new IllegalStateException("Stack is empty");
String item = top.data;
top = top.next;
size--;
return item;
}
public String peek() {
if (isEmpty()) throw new IllegalStateException("Stack is empty");
return top.data;
}
```

```
public boolean isEmpty() {
return top == null;
}
public int size() {
return size;
}
public Node getTop() {
return top;
}}
```

8) GameController Class:

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class GameController {
private MazeManager maze;
private Agent[] agents;
private int currentAgentIndex;
private int turnCount;
private int maxTurns;
private PlayerRegistry agentRegistry;
public GameController(int width, int height, int numAgents, int maxTurns) {
this.maze = new MazeManager(width, height, numAgents);
this.maze.initializeRotatingCorridors();
this.agents = maze.getAgents();
this.currentAgentIndex = 0;
this.turnCount = 0;
this.maxTurns = maxTurns;
this.agentRegistry = new PlayerRegistry();
for (Agent agent : agents) {
agentRegistry.register(agent);
}}
public MazeManager getMaze() {
return this.maze;
}
private boolean allAgentsFinished() {
for (Agent agent : agents) {
if (!agent.hasReachedGoal()) {
return false;
}}
return true;
}
private void printFinalResults() {
System.out.println("\n=== Game Over ===");
System.out.println("Total turns: " + turnCount);
List<Agent> goalAchievers = new ArrayList<>();
for (Agent agent : agents) {
if (agent.hasReachedGoal()) {
goalAchievers.add(agent);
}}
goalAchievers.sort(Comparator.comparingInt(Agent::getGoalReachedTurn));
System.out.println("\nGoal Achievement Order:");
if (goalAchievers.isEmpty()) {
```

```
System.out.println("No agents reached the goal!");
} else {
for (int i = 0; i < goalAchievers.size(); i++) {
Agent a = goalAchievers.get(i);
System.out.println((i+1) + ". Player " + (a.getId()+1) +
" (Turn " + a.getGoalReachedTurn() + ")");
}}
System.out.println("\nDetailed Statistics:");
System.out.println("+------+-------+-------+--------+------------+----------------+");
System.out.println("| Player | Moves | Traps | PwrUps | Backtracks | Goal Turn |");
System.out.println("+------+-------+-------+--------+------------+----------------+");
for (Agent agent : agents) {
System.out.printf("| %-6d | %-5d | %-5d | %-6d | %-10d | %-12s |\n",
agent.getId()+1,
agent.getTotalMoves(),
agent.getTrapsTriggered(),
agent.getPowerUpsCollected(),
agent.getBacktracks(),
agent.hasReachedGoal() ? agent.getGoalReachedTurn() : "DNF");
}
System.out.println("+------+-------+-------+--------+------------+----------------+");}
private void printAgentStatus(Agent agent) {
String colorCode = getAgentColor(agent.getId());
System.out.println(
colorCode + "Player " + (agent.getId()+1) + "\u001B[0m at (" +
agent.getCurrentX() + "," + agent.getCurrentY() + ")" +
" | Moves: " + agent.getTotalMoves() +
" | Backtracks: " + agent.getBacktracks()
);
System.out.print(" Power-ups: ");
if (agent.canTeleport()) System.out.print("Teleport, ");
if (agent.hasShield()) System.out.print("Shield, ");
if (agent.canDoubleMove()) System.out.print("Double Move, ");
if (!agent.hasPowerUp()) System.out.print("None");
System.out.println();
System.out.println(" Status: " + (agent.hasReachedGoal() ? "REACHED GOAL! 🏁" : "Still exploring"));
}
private String getAgentColor(int agentId) {
switch(agentId % 6) {
case 0: return "\u001B[31m"; // Red
case 1: return "\u001B[32m"; // Green
case 2: return "\u001B[33m"; // Yellow
case 3: return "\u001B[34m"; // Blue
case 4: return "\u001B[35m"; // Purple
case 5: return "\u001B[36m"; // Cyan
default: return "\u001B[37m"; // White
}}
public void startAutonomousGame() {
System.out.println("\n=== Game Starting ===");
System.out.println("Maze Size: " + maze.getWidth() + "x" + maze.getHeight());
System.out.println("Agent Number: " + agents.length);
System.out.println("Max Turns: " + maxTurns);
System.out.println("\nPower-Up Distribution:");
System.out.println("- Teleport (L): 25% ");
System.out.println("- Shield (S): 35% ");
System.out.println("- Double Move (D): 30% ");
System.out.println("- Time Extend (X): 10% ");
System.out.println("Goal Position: (" + maze.getGoalX() + "," + maze.getGoalY() + ")");
```

```
System.out.println("\n=== Initial Maze State ===");
maze.printMazeState();
maze.displayCorridorInfo();
AIController[] aiControllers = new AIController[agents.length];
for (int i = 0; i < agents.length; i++) {
aiControllers[i] = new AIController(maze, agents[i]);
}
while (turnCount < maxTurns && !allAgentsFinished()) {
turnCount++;
if (turnCount % 5 == 0) {
System.out.println("\n=== Turn " + turnCount + ": CORRIDOR ROTATION ===");
maze.rotateRandomCorridor();
maze.printMazeState();
try {
Thread.sleep(1500);
} catch (InterruptedException e) {
e.printStackTrace();
}
} else {
System.out.println("\n=== Turn " + turnCount + " ===");
}
Agent currentAgent = agents[currentAgentIndex];
if (!currentAgent.hasReachedGoal()) {
String move = aiControllers[currentAgentIndex].decideSmartMove();
if (move.equals("BACKTRACK")) {
currentAgent.backtrack();
System.out.println("Agent " + (currentAgent.getId()+1) + " backtracked!");
} else {
processAIMove(currentAgent, move);
}}
maze.printMazeState();
printAgentStatus(currentAgent);
do {
currentAgentIndex = (currentAgentIndex + 1) % agents.length;
} while (agents[currentAgentIndex].hasReachedGoal() && !allAgentsFinished());
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}}
printFinalResults();
logGameSummaryToFile("game_summary.txt");
}
private void processAIMove(Agent agent, String move) {
int newX = agent.getCurrentX();
int newY = agent.getCurrentY();
switch (move) {
case "UP": newY--; break;
case "DOWN": newY++; break;
case "LEFT": newX--; break;
case "RIGHT": newX++; break;
default: return;
}
if (maze.getGrid()[newX][newY].isTraversable()) {
agent.move(move);
maze.updateAgentPosition(agent, newX, newY);
System.out.println("Agent " + (agent.getId()+1) + " moved " + move);
maze.checkTileEffects(agent);
```

```
if (agent.canDoubleMove()) {
System.out.println("Agent " + (agent.getId()+1) + " uses double move!");
newX = agent.getCurrentX();
newY = agent.getCurrentY();
switch (move) {
case "UP": newY--; break;
case "DOWN": newY++; break;
case "LEFT": newX--; break;
case "RIGHT": newX++; break;
}
if (newX >= 0 && newX < maze.getWidth() &&
newY >= 0 && newY < maze.getHeight() &&
maze.getGrid()[newX][newY].isTraversable()) {
agent.move(move);
maze.updateAgentPosition(agent, newX, newY);
System.out.println("Agent " + (agent.getId()+1) + " used second move: " + move);
maze.checkTileEffects(agent);
}}
if (agent.canTeleport() && Math.random() < 0.3) {
performTeleport(agent);
}}
agent.decrementPowerUpEffects();
}
private void performTeleport(Agent agent) {
if (!agent.canTeleport()) return;
agent.useTeleport();
System.out.println("Agent " + (agent.getId()+1) + " using teleport!");
int newX, newY;
int attempts = 0;
do {
newX = 1 + (int)(Math.random() * (maze.getWidth() - 2));
newY = 1 + (int)(Math.random() * (maze.getHeight() - 2));
attempts++;
} while (!maze.getGrid()[newX][newY].isTraversable() && attempts < 50);
if (attempts < 50) {
maze.getGrid()[agent.getCurrentX()][agent.getCurrentY()].clearAgent();
maze.updateAgentPosition(agent, newX, newY);
System.out.println("Agent " + (agent.getId()+1) + " teleported to (" + newX + "," + newY + ")");
maze.checkTileEffects(agent);
} else {
System.out.println("Agent " + (agent.getId()+1) + " failed to teleport!");
}}
public void logGameSummaryToFile(String filename) {
try (FileWriter writer = new FileWriter(filename)) {
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
writer.write("=== Maze Escape Game Summary ===\n");
writer.write("Generated on: " + dateFormat.format(new Date()) + "\n\n");
writer.write("Game Configuration:\n");
writer.write(String.format("- Maze Size: %dx%d\n", maze.getWidth(), maze.getHeight()));
writer.write(String.format("- Number of Agents: %d\n", agents.length));
writer.write(String.format("- Max Turns: %d\n", maxTurns));
writer.write(String.format("- Goal Position: (%d,%d)\n\n", maze.getGoalX(), maze.getGoalY()));
writer.write("Game Outcome:\n");
writer.write(String.format("- Total Turns Played: %d\n", turnCount));
List<Agent> goalAchievers = new ArrayList<>();
for (Agent agent : agents) {
if (agent.hasReachedGoal()) {
goalAchievers.add(agent);
```

```
}}
goalAchievers.sort(Comparator.comparingInt(Agent::getGoalReachedTurn));
writer.write("\nGoal Achievement Order:\n");
if (goalAchievers.isEmpty()) {
writer.write("No agents reached the goal!\n");
} else {
for (int i = 0; i < goalAchievers.size(); i++) {
Agent a = goalAchievers.get(i);
writer.write(String.format("%d. Player %d (Turn %d)\n",
i+1, a.getId()+1, a.getGoalReachedTurn()));
}}
writer.write("\nDetailed Statistics:\n");
writer.write("+--------+-------+-------+--------+------------+-------------+\n");
writer.write("| Player | Moves | Traps | PwrUps | Backtracks | Goal Reached |\n");
writer.write("+--------+-------+-------+--------+------------+-------------+\n");
for (Agent agent : agents) {
writer.write(String.format("| %-6d | %-5d | %-5d | %-6d | %-10d | %-11s |\n",
agent.getId()+1,
agent.getTotalMoves(),
agent.getTrapsTriggered(),
agent.getPowerUpsCollected(),
agent.getBacktracks(),
agent.hasReachedGoal() ? "Yes" : "No"));
}
writer.write("+--------+-------+-------+--------+------------+-------------+\n\n");
writer.write("\n=== End of Summary ===\n");
System.out.println("Game summary successfully written to " + filename);
} catch (IOException e) {
System.err.println("Error writing game summary to file: " + e.getMessage());
e.printStackTrace();
}}}
```

9) MazeManager Class:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class MazeManager {
private MazeTile[][] grid;
private int width, height;
private Agent[] agents;
private int goalX, goalY;
private static final int[][] DIRECTIONS = {
{0, -1}, // UP
{1, 0}, // RIGHT
{0, 1}, // DOWN
{-1, 0} // LEFT
};
public MazeManager(int width, int height, int numAgents) {
this.width = width;
this.height = height;
this.grid = new MazeTile[width][height];
this.agents = new Agent[numAgents];
generatePerfectMaze();
placeSpecialTiles();
```

```
initializeAgents(numAgents);
initializeRotatingCorridors();
}
private void generatePerfectMaze() {
for (int x = 0; x < width; x++) {
for (int y = 0; y < height; y++) {
if (x == 0 || y == 0 || x == width-1 || y == height-1) {
grid[x][y] = new MazeTile(x, y, 'W');
} else {
grid[x][y] = new MazeTile(x, y, 'W');
}}}
Stack stack = new Stack();
int startX = 1;
int startY = 1;
grid[startX][startY].setType('E');
stack.push(new int[]{startX, startY});
while (!stack.isEmpty()) {
int[] current = stack.peek();
int x = current[0];
int y = current[1];
CustomList neighbors = new CustomList();
for (int[] dir : DIRECTIONS) {
int nx = x + 2 * dir[0];
int ny = y + 2 * dir[1];
if (nx >= 1 && nx <= width-2 && ny >= 1 && ny <= height-2 &&
grid[nx][ny].getType() == 'W') {
neighbors.add(new int[]{nx, ny, x + dir[0], y + dir[1]});
}}
if (!neighbors.isEmpty()) {
int[] chosen = neighbors.removeRandom();
int nx = chosen[0];
int ny = chosen[1];
int wallX = chosen[2];
int wallY = chosen[3];
grid[nx][ny].setType('E');
grid[wallX][wallY].setType('E');
stack.push(new int[]{nx, ny});
} else {
stack.pop();
}}
for (int x = 1; x < width-1; x++) {
if (grid[x][height-3].getType() == 'E') {
grid[x][height-2].setType('E');
}}
for (int y = 1; y < height-1; y++) {
if (grid[width-3][y].getType() == 'E') {
grid[width-2][y].setType('E');
}}
goalX = width - 2;
goalY = height - 2;
grid[goalX][goalY].setType('G');
}
private void placeSpecialTiles() {
int trapCount = (int)(countEmptyTiles() * 0.1);
while (trapCount > 0) {
int x = 1 + (int)(Math.random() * (width - 2));
int y = 1 + (int)(Math.random() * (height - 2));
if (grid[x][y].getType() == MazeTile.EMPTY && !(x == goalX && y == goalY)) {
```

```
grid[x][y].setType(MazeTile.TRAP);
trapCount--;
}}
char[] powerUpTypes = new char[100];
Arrays.fill(powerUpTypes, 0, 25, MazeTile.POWERUP_TELEPORT);
Arrays.fill(powerUpTypes, 25, 60, MazeTile.POWERUP_SHIELD);
Arrays.fill(powerUpTypes, 60, 90, MazeTile.POWERUP_DOUBLE_MOVE);
Arrays.fill(powerUpTypes, 90, 100, MazeTile.POWERUP_TIME_EXTEND);
int powerUpCount = (int)(countEmptyTiles() * 0.15);
while (powerUpCount > 0) {
int x = 1 + (int)(Math.random() * (width - 2));
int y = 1 + (int)(Math.random() * (height - 2));
if (grid[x][y].getType() == 'E' && !(x == goalX && y == goalY)) {
char randomType = powerUpTypes[(int)(Math.random() * powerUpTypes.length)];
grid[x][y].setType(randomType);
powerUpCount--;
}}}
private int countEmptyTiles() {
int count = 0;
for (int x = 0; x < width; x++) {
for (int y = 0; y < height; y++) {
if (grid[x][y].getType() == 'E') count++;
}}
return count;
}
private void initializeAgents(int numAgents) {
List<int[]> validStartPositions = new ArrayList<>();
for (int x = 1; x < width-1; x++) {
for (int y = 1; y < height-1; y++) {
if (grid[x][y].getType() == 'E' && !(x == goalX && y == goalY)) {
validStartPositions.add(new int[]{x, y});
}}}
Collections.shuffle(validStartPositions);
for (int i = 0; i < numAgents && i < validStartPositions.size(); i++) {
int[] pos = validStartPositions.get(i);
agents[i] = new Agent(i, pos[0], pos[1]);
grid[pos[0]][pos[1]].setAgent(i);
}}
public void printMazeState() {
System.out.println("\nCurrent Maze State:");
System.out.print("  ┌");
for (int x = 0; x < width; x++) System.out.print("──");
System.out.println("┐ ");
for (int y = 0; y < height; y++) {
System.out.print(y + " │");
for (int x = 0; x < width; x++) {
System.out.print(" " + grid[x][y]);
}
System.out.println(" │");
}
System.out.print("  └");
for (int x = 0; x < width; x++) System.out.print("──");
System.out.println("┘ ");
System.out.print("  ");
for (int x = 0; x < width; x++) {
System.out.print(x + " ");
}
System.out.println();
```

```
System.out.println("\nAgent Status:");
for (Agent agent : agents) {
int displayId = agent.getId() + 1;
String colorCode = getAgentColor(agent.getId());
System.out.println(colorCode + "Agent " + displayId + "\u001B[0m: " +
"At (" + agent.getCurrentX() + "," + agent.getCurrentY() + ") " +
(agent.hasPowerUp() ? " ⚡ " : "") +
(agent.hasReachedGoal() ? " 🏁 " : ""));
}}
public boolean isValidMove(int fromX, int fromY, String direction) {
int toX = fromX, toY = fromY;
switch (direction) {
case "UP": toY--; break;
case "DOWN": toY++; break;
case "LEFT": toX--; break;
case "RIGHT": toX++; break;
default: return false;
}
if (toX < 0 || toX >= width || toY < 0 || toY >= height) {
return false;
}
return grid[toX][toY].isTraversable();
}
public String getAgentColor(int agentId) {
switch(agentId % 6) {
case 0: return "\u001B[31m"; // Red
case 1: return "\u001B[32m"; // Green
case 2: return "\u001B[33m"; // Yellow
case 3: return "\u001B[34m"; // Blue
case 4: return "\u001B[35m"; // Purple
case 5: return "\u001B[36m"; // Cyan
default: return "\u001B[37m"; // White
}}
public void updateAgentPosition(Agent agent, int newX, int newY) {
int oldX = agent.getCurrentX();
int oldY = agent.getCurrentY();
grid[oldX][oldY].clearAgent();
agent.setCurrentX(newX);
agent.setCurrentY(newY);
grid[newX][newY].setAgent(agent.getId());
}
public void checkTileEffects(Agent agent) {
MazeTile currentTile = grid[agent.getCurrentX()][agent.getCurrentY()];
if (agent.hasReachedGoal()) {
return;}
if (currentTile.isTrap()) {
System.out.println("Agent " + (agent.getId()+1) + " hit a trap!");
agent.incrementTrapsTriggered();
currentTile.clearAgent();
if (!agent.hasShield()) {
agent.backtrack();
MazeTile backtrackTile = grid[agent.getCurrentX()][agent.getCurrentY()];
backtrackTile.setAgent(agent.getId());
System.out.println("Agent " + (agent.getId()+1) + " backtracked to (" +
agent.getCurrentX() + "," + agent.getCurrentY() + ")");
currentTile.setType(MazeTile.EMPTY);
} else {
System.out.println("Agent " + (agent.getId()+1) + "'s shield protected from trap!");
```

```
currentTile.setType(MazeTile.EMPTY);
}
}
else if (currentTile.isPowerUp()) {
agent.incrementPowerUpsCollected();
System.out.println("Agent " + (agent.getId()+1) + " found a power-up!");
char powerUpType = currentTile.getType();
agent.applyPowerUp(powerUpType);
currentTile.setType(MazeTile.EMPTY);
}
if (grid[agent.getCurrentX()][agent.getCurrentY()].getType() == MazeTile.GOAL) {
agent.setReachedGoal(true);
agent.setGoalReachedTurn(agent.getTotalMoves());
System.out.println("Agent " + (agent.getId()+1) + " reached the goal!");
}}
public Agent[] getAgents() {
return agents;
}
public int getGoalX() { return goalX; }
public int getGoalY() { return goalY; }
public int getWidth() {
return width;
}
public int getHeight() {
return height;
}
private List<CircularCorridor> rotatingCorridors;
private int rotationInterval = 5;
public void initializeRotatingCorridors() {
rotatingCorridors = new ArrayList<>();
if (width >= 5 && height >= 5) {
addHorizontalCorridor(height / 3);
addHorizontalCorridor((height * 2) / 3);
addVerticalCorridor(width / 3);
addVerticalCorridor((width * 2) / 3);
if (width >= 7 && height >= 7) {
addSquareCorridor(2, 2); // Top-left
addSquareCorridor(width - 4, 2); // Top-right
addSquareCorridor(2, height - 4); // Bottom-left
addSquareCorridor(width - 4, height - 4); // Bottom-right
}}
System.out.println("Initialized " + rotatingCorridors.size() + " rotating corridors");
}
private void addHorizontalCorridor(int row) {
if (row <= 0 || row >= height - 1) return;
MazeTile[] tiles = new MazeTile[width];
for (int x = 0; x < width; x++) {
tiles[x] = grid[x][row];
}
rotatingCorridors.add(new CircularCorridor(tiles, row, -1, CircularCorridor.CorridorType.HORIZONTAL));
System.out.println("Added horizontal corridor at row " + row);
}
private void addVerticalCorridor(int col) {
if (col <= 0 || col >= width - 1) return;
MazeTile[] tiles = new MazeTile[height];
for (int y = 0; y < height; y++) {
tiles[y] = grid[col][y];
}
```

```
rotatingCorridors.add(new CircularCorridor(tiles, -1, col, CircularCorridor.CorridorType.VERTICAL));
System.out.println("Added vertical corridor at column " + col);
}
private void addSquareCorridor(int x, int y) {
if (x <= 0 || y <= 0 || x + 1 >= width - 1 || y + 1 >= height - 1) return;
if (!grid[x][y].isTraversable() || !grid[x+1][y].isTraversable() ||
!grid[x][y+1].isTraversable() || !grid[x+1][y+1].isTraversable()) {
return;
}
MazeTile[] tiles = {
grid[x][y], // Top-left
grid[x+1][y], // Top-right
grid[x+1][y+1], // Bottom-right
grid[x][y+1] // Bottom-left
};
rotatingCorridors.add(new CircularCorridor(tiles, x, y, CircularCorridor.CorridorType.SQUARE));
System.out.println("Added square corridor at position (" + x + "," + y + ")");
}
public void checkForRotation(int turnCount) {
if (turnCount % rotationInterval == 0) {
rotateRandomCorridor();
}}
public void rotateRandomCorridor() {
if (rotatingCorridors.isEmpty()) {
System.out.println("No corridors available to rotate!");
return;
}
int index = (int)(Math.random() * rotatingCorridors.size());
CircularCorridor corridor = rotatingCorridors.get(index);
List<int[]> agentPositions = new ArrayList<>();
for (Agent agent : agents) {
agentPositions.add(new int[]{agent.getId(), agent.getCurrentX(), agent.getCurrentY()});
}
corridor.rotate();
updateGridAfterRotation(corridor);
updateAgentPositionsAfterRotation(agentPositions);
if (corridor.isHorizontal()) {
System.out.println("Rotated horizontal corridor at row " + corridor.getRow());
} else if (corridor.isVertical()) {
System.out.println("Rotated vertical corridor at column " + corridor.getCol());
} else {
System.out.println("Rotated square corridor at (" + corridor.getX() + "," + corridor.getY() + ")");
}}
private void updateGridAfterRotation(CircularCorridor corridor) {
MazeTile[] tiles = corridor.getTiles();
if (corridor.isHorizontal()) {
int row = corridor.getRow();
for (int x = 0; x < width; x++) {
grid[x][row] = tiles[x];
}
} else if (corridor.isVertical()) {
int col = corridor.getCol();
for (int y = 0; y < height; y++) {
grid[col][y] = tiles[y];
}
} else if (corridor.isSquare()) {
int x = corridor.getX();
int y = corridor.getY();
```

```
grid[x][y] = tiles[0]; // Top-left
grid[x+1][y] = tiles[1]; // Top-right
grid[x+1][y+1] = tiles[2]; // Bottom-right
grid[x][y+1] = tiles[3]; // Bottom-left
}}
public void testCorridorRotation() {
initializeRotatingCorridors();
System.out.println("Before rotation:");
printMazeState();
for (CircularCorridor corridor : rotatingCorridors) {
System.out.println("\nRotating " +
(corridor.isHorizontal() ? "horizontal" :
corridor.isVertical() ? "vertical" : "square") + " corridor");
corridor.rotate();
updateGridAfterRotation(corridor);
printMazeState();
}}
public void displayCorridorInfo() {
if (rotatingCorridors == null || rotatingCorridors.isEmpty()) {
System.out.println("There is no rotating corridor.");
return;
}
System.out.println("\n=== Rotating Corridors ===");
System.out.println("Total number of rotating corridors: " + rotatingCorridors.size());
for (int i = 0; i < rotatingCorridors.size(); i++) {
CircularCorridor corridor = rotatingCorridors.get(i);
System.out.print((i+1) + ". ");
if (corridor.isHorizontal()) {
System.out.println("Horizontal Corridor - Row " + corridor.getRow());
} else if (corridor.isVertical()) {
System.out.println("Vertical Corridor - Column" + corridor.getCol());
} else {
System.out.println("Square Block - Location (" + corridor.getX() + "," + corridor.getY() + ")");
}}}
public MazeTile[][] getGrid() {
return grid;
}
public MazeTile getGridTile(int x, int y) {
if (x >= 0 && x < width && y >= 0 && y < height) {
return grid[x][y];
}
return null;
}
public MazeTile getTile(int x, int y) {
if (x >= 0 && x < width && y >= 0 && y < height) {
return grid[x][y];}
return null;}
private void updateAgentPositionsAfterRotation(List<int[]> previousPositions) {
for (int[] pos : previousPositions) {
int agentId = pos[0];
int oldX = pos[1];
int oldY = pos[2];
Agent agent = agents[agentId];if (!grid[oldX][oldY].hasAgent() || grid[oldX][oldY].getAgentId() != agentId) {
boolean found = false;
for (int x = 0; x < width && !found; x++) {
for (int y = 0; y < height && !found; y++) {
if (grid[x][y].hasAgent() && grid[x][y].getAgentId() == agentId) {
agent.setCurrentX(x);
```

```
agent.setCurrentY(y);
found = true;
System.out.println("Agent " + (agentId+1) + " moved from (" +
oldX + "," + oldY + ") to (" + x + "," + y + ") due to corridor rotation");
}}}
if (!found) {
grid[oldX][oldY].setAgent(agentId);
System.out.println("Agent " + (agentId+1) + " restored to original position");
}}}}}
```

10) MazeTile Class:

```
public class MazeTile {
private int x, y;
private char type; // 'E':Empty, 'W':Wall, 'T':Trap, 'P':Power-up, 'G':Goal
private boolean hasAgent;
private int agentId;
public static final char EMPTY = 'E';
public static final char WALL = 'W';
public static final char TRAP = 'T';
public static final char GOAL = 'G';
public static final char POWERUP_TELEPORT = 'L'; // 'T' trap ile çakışmaması için
public static final char POWERUP_SHIELD = 'S';
public static final char POWERUP_DOUBLE_MOVE = 'D';
public static final char POWERUP_TIME_EXTEND = 'X';
public MazeTile(int x, int y, char type) {
this.x = x;
this.y = y;
this.type = type;
this.hasAgent = false;
this.agentId=-1;
}
public boolean isTraversable() {
return type != 'W' || type == 'G';
}
public char getType() {
return type;
}
public void setType(char type) {
this.type = type;
}
public void setHasAgent(boolean hasAgent) {
this.hasAgent = hasAgent;
}
public boolean hasAgent() {
return hasAgent;
}
public int getX() { return x; }
public int getY() { return y; }
@Override
public String toString() {
if (hasAgent) {
int displayId = Math.max(0, agentId) + 1;
switch(displayId % 6) {
case 1: return "\u001B[31m" + displayId + "\u001B[0m"; // Red 1
case 2: return "\u001B[32m" + displayId + "\u001B[0m"; // Green 2
case 3: return "\u001B[33m" + displayId + "\u001B[0m"; // Yellow 3
case 4: return "\u001B[34m" + displayId + "\u001B[0m"; // Blue 4
```

```
case 5: return "\u001B[35m" + displayId + "\u001B[0m"; // Purple 5
case 0: return "\u001B[36m6\u001B[0m"; // Cyan 6
default: return "\u001B[37m?\u001B[0m"; // White ?
}}
switch (type) {
case WALL: return "\u001B[37m■\u001B[0m";
case EMPTY: return " ";
case TRAP: return "\u001B[31mX\u001B[0m";
case POWERUP_TELEPORT: return "\u001B[35m⊠\u001B[0m";
case POWERUP_SHIELD: return "\u001B[34m⛨\u001B[0m";
case POWERUP_DOUBLE_MOVE: return "\u001B[33m↝\u001B[0m";
case POWERUP_TIME_EXTEND: return "\u001B[36m⏱\u001B[0m";
case GOAL: return "\u001B[32m◉\u001B[0m";
default: return "?";
}}
public void setAgent(int agentId) {
this.hasAgent = true;
this.agentId = agentId;
}
public void clearAgent() {
this.hasAgent = false;
this.agentId = -1;
}
public int getAgentId() {
return agentId;
}
public boolean isPowerUp() {
return type == POWERUP_TELEPORT ||
type == POWERUP_SHIELD ||
type == POWERUP_DOUBLE_MOVE ||
type == POWERUP_TIME_EXTEND;
}
public boolean isTrap() {
return type == 'T';
}}
```

11) PlayerRegistry Class:

```
public class PlayerRegistry {
private static class Node {
Agent data;
Node next;
Node(Agent data) {
this.data = data;
this.next = null;
}
}
private Node head;
private int size;
public void register(Agent agent) {
Node newNode = new Node(agent);
if (head == null) {
head = newNode;
} else {
Node current = head;
while (current.next != null) {
current = current.next;
}
```

```
current.next = newNode;
}
size++;
}
public Agent getAgent(int index) {
if (index < 0 || index >= size) return null;
Node current = head;
for (int i = 0; i < index; i++) {
current = current.next;
}
return current.data;
}}
```

12) Stack Class:

```
class Stack {
private Node top;
private int size;
private static class Node {
int[] data;
Node next;
Node(int[] data) {
this.data = data;
this.next = null;
}}
public Stack() {
top = null;
size = 0;}
public void push(int[] item) {
Node newNode = new Node(item);
newNode.next = top;
top = newNode;
size++;
}
public int[] pop() {
if (isEmpty()) throw new IllegalStateException("Stack is empty");
int[] item = top.data;
top = top.next;
size--;
return item;
}
public int[] peek() {
if (isEmpty()) throw new IllegalStateException("Stack is empty");
return top.data;
}
public boolean isEmpty() {
return top == null;
}
public int size() {
return size;
}}
```

13) TurnManager Class:

```
public class TurnManager {
private CustomQueue agentQueue;
private int currentRound;
```

```
private StringBuilder gameLog;
public TurnManager(Agent[] agents) {
this.agentQueue = new CustomQueue();
this.currentRound = 0;
this.gameLog = new StringBuilder();
for (Agent agent : agents) {
agentQueue.enqueue(agent);
}}
public void advanceTurn() {
currentRound++;
rotateAgents();}
private void rotateAgents() {
if (!agentQueue.isEmpty()) {
Agent first = agentQueue.dequeue();
agentQueue.enqueue(first);
}}
public Agent getCurrentAgent() {
return agentQueue.peek();}
public boolean allAgentsFinished() {
return agentQueue.allAgentsFinished(); // Delegate to CustomQueue's method}
public void logTurnSummary(Agent a, String action) {
String logEntry = String.format("Turn %d - Agent %d: %s\n",
currentRound, a.getId(), action);
gameLog.append(logEntry);
System.out.print(logEntry);}
public String getGameLog() {
return gameLog.toString();}
public int getCurrentRound() {
return currentRound;
}}
```

Terminology

Game Concepts

Perfect Maze: A maze that has exactly one path between any two points, with no loops or isolated sections. Every cell in a perfect maze is reachable from any other cell through exactly one unique path.

Rotating Corridor: A section of the maze (horizontal, vertical, or square arrangement of tiles) that can rotate during gameplay, changing the maze configuration and pathways.

Turn-Based Movement: The game mechanic where agents take turns making moves rather than moving simultaneously, managed by the TurnManager.

Autonomous Game: A game mode where all agents are controlled by AI rather than human players, making decisions based on pathfinding algorithms.

Round-Robin Scheduling: A method of giving turns to agents in a circular order, ensuring each agent gets equal opportunity to move.

Tile Types

Empty Tile: A traversable space in the maze where agents can freely move.

Wall: An impassable obstacle that blocks agent movement.

Trap: A hazardous tile that negatively affects agents who step on it, unless protected by a shield.

Goal: The destination tile that agents must reach to complete the maze.

Power-Up: Special tiles that grant beneficial effects to agents who collect them.

Data Structure Terminology

Node: The fundamental unit in linked data structures, containing a data element and reference(s) to other node(s).

LIFO (Last-In-First-Out): The operational principle of stacks, where the most recently added element is the first to be removed.

FIFO (First-In-First-Out): The operational principle of queues, where the earliest added element is the first to be removed.

Traversal: The process of visiting every node in a data structure, typically used in searching or modifying operations.

Backtracking: An algorithmic technique where the solution is built incrementally, abandoning a partial solution ("backtracking") when it determines the solution cannot be completed.

Agent Terms

Backtrack: When an agent retraces its steps, moving in the reverse direction of a previous move.

Move History: A record of all moves an agent has made, stored as directions in a stack data structure.

Agent ID: A unique identifier assigned to each agent, used for tracking and distinguishing between multiple agents.

Agent Status: The current state of an agent including position, power-up effects, and goal achievement.

Algorithm Terminology

Depth-First Search (DFS): An algorithm for traversing or searching tree or graph data structures, exploring as far as possible along each branch before backtracking.

Breadth-First Search (BFS): An algorithm for traversing or searching tree or graph data structures, exploring all neighbor nodes at the present depth before moving to nodes at the next depth level.

Pathfinding: The process of finding the shortest or optimal path between two points, commonly used by AI agents to navigate the maze.

Heuristic: A technique used in pathfinding algorithms to make informed decisions about which path to explore next, based on an estimate of the remaining distance to the goal.