

# Dokumentation des Labyrinthspiels

## I. Wie das Spiel funktioniert

Das Labyrinthspiel ist eine Java-basierte Anwendung, die Spieler herausfordert, sich durch ein zufällig generiertes Labyrinth zu navigieren, um den Ausgang zu erreichen. Das Spiel bietet zwei Hauptmodi: **Einzelspieler** und **Wettkampfmodus**. Es beinhaltet Funktionen wie Labyrinthgenerierung, KI-Lösung, Labyrinthregeneration, Speichern/Laden von Spielständen und visuelles Rendering mit Turtle.

### Spielmodi

#### 1. Single Player Mode:

- Der Spieler navigiert durch das Labyrinth, um den Ausgang zu erreichen.
- Der Spieler kann Hinweise (nächste 10 Schritte) oder die vollständige Lösung anfordern.
- Das Spiel verfolgt die Züge des Spielers und die verstrichene Zeit.

#### 2. Competitive Mode:

- Der Spieler tritt gegen eine KI an, um zuerst den Ausgang zu erreichen.
- Spieler wechseln sich ab, Züge zu machen, und das Spiel verfolgt die Züge sowohl des Spielers als auch der KI.
- Die KI folgt einem vorab berechneten optimalen Pfad zum Ausgang.

### Features

- **Maze Generation:** Das Labyrinth wird mit einem Depth-First Search (DFS)-Algorithmus generiert, was jedes Mal ein einzigartiges und lösbares Labyrinth sicherstellt.
- **AI Solver:** Die KI verwendet einen DFS-basierten Algorithmus, um den optimalen Pfad vom Eingang zum Ausgang zu finden.
- **Maze Regeneration:** Das Labyrinth regeneriert sich alle 60 Sekunden und bewahrt die Position des Spielers, wenn möglich.
- **Save/Load Game:** Spieler können ihren Fortschritt speichern und später laden, um das Spiel fortzusetzen.
- **Visual Rendering:** Das Labyrinth wird mit einer Turtle-Grafikbibliothek gezeichnet, mit Farben für Wände, Spieler, Eingang, Ausgang und Lösungspfade.
- **Timer and Move Counter:** Das Spiel zeigt die verstrichene Zeit und die Anzahl der vom Spieler (und der KI im Wettkampfmodus) gemachten Züge an.

### Winning Conditions

- **Single Player Mode:** Der Spieler gewinnt, indem er den Ausgang erreicht. Das Spiel zeigt die verstrichene Zeit und die Anzahl der Züge an.
- **Competitive Mode:** Der Spieler oder die KI gewinnt, indem sie zuerst den Ausgang erreichen. Das Spiel zeigt die Anzahl der Züge beider Spieler und den Gewinner an.

## II. Codeübersicht

### 1. Klassenübersicht

#### Maze-Klasse

Die `Maze`-Klasse ist die zentrale Komponente des Labyrinthspiels und verantwortlich für die Verwaltung der Labyrinthgenerierung, Spielerbewegungen, KI-Interaktionen, Spielmodi, das Speichern/Laden von Spielständen und das Rendern. Sie nutzt andere unterstützende Klassen wie `Cell`, `Player` und `Point`, um verschiedene Aspekte des Spiels zu kapseln.

#### Hauptverantwortlichkeiten:

- **Maze Generation:** Erstellt ein einzigartiges, lösbares Labyrinth mit dem Depth-First Search (DFS)-Algorithmus.
  - **AI Solving:** Implementiert eine KI, die das Labyrinth optimal lösen kann.
  - **Spielverwaltung:** Handhabt Spielmodi, Spielturniere und Spielzustandsübergänge.
  - **Rendering:** Verwendet eine Turtle-Grafikbibliothek, um das Labyrinth, die Spieler, die KI und Lösungspfade visuell darzustellen.
  - **Persistenz:** Ermöglicht das Speichern und Laden von Spielständen für fortgesetztes Spielen.
- 

### 2. Konstanten

Die `Maze`-Klasse definiert mehrere Konstanten zur Konfiguration der Spieleinstellungen und Rendering-Eigenschaften:

- **Labyrinthabmessungen:**
    - `MIN_SIZE`: Mindestgröße für die Labyrinthabmessungen.
    - `MIN_CELL_SIZE`: Mindestpixelgröße für jede Zelle im Labyrinth.
  - **Farben (RGB):**
    - `PLAYER_COLOR`: Farbe, die den menschlichen Spieler darstellt (Blau).
    - `AI_COLOR`: Farbe, die den KI-Spieler darstellt (Rot).
    - `WALL_COLOR`: Farbe der Labyrinthwände (Schwarz).
    - `ENTRANCE_COLOR`: Farbe des Labyrintheingangs (Grün).
    - `EXIT_COLOR`: Farbe des Labyrinthausgangs (Rot).
    - `TIMER_TEXT_COLOR`: Farbe des Timers und der Zugzähler (Schwarz).
    - `PATH_COLOR`: Farbe zur Darstellung von Lösungspfaden (Grün).
  - **Timer:**
    - `REGENERATION_TIME`: Intervall (in Sekunden), in dem das Labyrinth automatisch regeneriert wird.
- 

### 3. Felder

Die `Maze`-Klasse verwaltet die folgenden Felder zur Verwaltung des Spielzustands:

- **Labyrinthabmessungen:**

- width: Anzahl der Zellen horizontal.
- height: Anzahl der Zellen vertikal.
- cellSize: Pixelgröße jeder Zelle für das Rendering.

- **Labyrinthdaten:**

- grid: 2D-Array von Cell-Objekten, die die Labyrinthstruktur darstellen.
- random: Instanz von Random zur Erzeugung zufälliger Zahlen während der Labyrinthherstellung.

- **Eingang und Ausgang:**

- entrance: Cell-Objekt, das den Eingang des Labyrinths darstellt.
- exit: Cell-Objekt, das den Ausgang des Labyrinths darstellt.

- **Spieler:**

- player: Instanz von Player, die den menschlichen Spieler darstellt.
- aiPlayer: Instanz von Player, die den KI-Spieler im Wettkampfmodus darstellt.

- **AI Solution:**

- aiSolution: Liste von Character, die den vollständigen Lösungspfad der KI darstellt.
- aiPlannedMoves: Liste von Character, die die geplanten Züge der KI darstellt.

- **Rendering:**

- turtle: Instanz der Turtle-Grafikbibliothek, die zum Zeichnen des Labyrinths und der Spieler verwendet wird.

- **Timer:**

- gameTimer: Timer-Objekt, das die verstrichene Spielzeit verfolgt.
- displayTimer: Timer-Objekt, das möglicherweise für periodische Updates verwendet wird (im Code nicht detailliert).
- regenerationTimer: Timer-Objekt, das die Labyrinthregeneration in festgelegten Intervallen auslöst.

- **Spielzustand:**

- elapsedTime: Gesamte verstrichene Zeit seit Spielbeginn in Sekunden.
- isGameRunning: Boolean-Flag, das anzeigt, ob das Spiel derzeit aktiv ist.
- isRegenerating: Boolean-Flag, das anzeigt, ob eine Labyrinthregeneration im Gange ist.

- **Wettkampfmodus:**

- isCompetitiveMode: Boolean-Flag, das anzeigt, ob das Spiel im Wettkampfmodus ist.
- humanMoveCount: Zähler für die Anzahl der vom menschlichen Spieler gemachten Züge.
- aiMoveCount: Zähler für die Anzahl der von der KI gemachten Züge.
- isHumanTurn: Boolean-Flag, das anzeigt, ob gerade der menschliche Spieler im Wettkampfmodus am Zug ist.

- **Input Handling:**

- scanner: Scanner-Instanz zum Lesen von Benutzereingaben aus der Konsole.

## 4. Konstruktor

```
Maze(Turtle turtle, int width, int height, int cellSize)
```

- **Zweck:** Initialisiert eine neue Maze-Instanz mit den angegebenen Abmessungen und Rendering-Einstellungen.
- **Parameter:**

- `turtle`: Die Turtle-Instanz, die zum Rendern des Labyrinths und der Spieler verwendet wird.
- `width`: Die Breite des Labyrinths in Zellen.
- `height`: Die Höhe des Labyrinths in Zellen.
- `cellSize`: Die Pixelgröße jeder Zelle.

- **Prozess:**

1. Weist die Labyrinthabmessungen zu und validiert sie, um sicherzustellen, dass sie die Mindestgrößenanforderungen erfüllen.
  2. Initialisiert das Labyrinthgitter durch Aufruf von `initializeGrid()`.
  3. Initialisiert die KI-Lösungslisten (`aiSolution` und `aiPlannedMoves`).
  4. Setzt die Wettkampfmodus-Flags und Zugzähler auf Standardwerte.
- 

## 5. Hauptmethoden

Hauptmethoden sind die Kernfunktionen, die die Hauptmerkmale des Spiels antreiben. Sie rufen oft Hilfsmethoden auf, um spezifische Aufgaben auszuführen.

### 5.1. Labyrinthgenerierung

```
generate()
```

- **Zweck:** Generiert ein neues Labyrinthlayout mithilfe des Depth-First Search (DFS)-Algorithmus.
  - **Prozess:**
1. Ruft `resetGrid()` auf, um das Labyrinthgitter zu initialisieren.
  2. Startet DFS von der Anfangszelle (`grid[0][0]`), markiert sie als besucht.
  3. Verwendet ein `stack` (`List<Cell>`), um Zellen zu durchlaufen und Pfade zu erstellen, indem Wände zwischen der aktuellen und der nächsten Zelle entfernt werden.
  4. Setzt fort, bis alle Zellen besucht sind.
  5. Ruft `createEntranceAndExit()` auf, um die Eingang- und Ausgangspunkte des Labyrinths zu definieren.

### 5.2. KI-Lösung

```
findSolution(Cell current, Set<Cell> visited, List<Character> currentPath)
```

- **Zweck:** Sucht rekursiv nach einem Pfad von der aktuellen Zelle zum Ausgang unter Verwendung von DFS.
  - **Prozess:**
1. Markiert die aktuelle Zelle als besucht.
  2. Wenn die aktuelle Zelle der Ausgang ist, fügt den aktuellen Pfad zu `aiSolution` hinzu.
  3. Iteriert durch alle möglichen Richtungen (Oben, Rechts, Unten, Links).

- 4. Für jede Richtung ohne Wand bewegt sich zur Nachbarzelle und setzt die Suche fort.
- 5. Rückverfolgung, wenn in einer Richtung kein Pfad gefunden wird.

`solveMaze()`

- **Zweck:** Initiiert den Labyrinthlösungsprozess und zeigt den vollständigen Lösungspfad visuell an.
- **Prozess:**
  1. Löscht vorhandene KI-Lösungen in `aiSolution`.
  2. Ruft die aktuelle Position des Spielers ab (`Cell`).
  3. Ruft `findSolution()` auf, um den Lösungspfad zu berechnen.
  4. Wenn eine Lösung gefunden wird, ruft `drawPathInGreen()` auf, um den Pfad zu visualisieren.
  5. Benachrichtigt den Spieler über die Anzeige der Lösung.

`provideNext10Steps()`

- **Zweck:** Zeigt die nächsten 10 Schritte des KI-Lösungspfads von der aktuellen Position des Spielers an.
- **Prozess:**
  1. Löscht vorhandene KI-Lösungen in `aiSolution`.
  2. Ruft die aktuelle Position des Spielers ab (`Cell`).
  3. Ruft `findSolution()` auf, um den Lösungspfad zu berechnen.
  4. Extrahiert die nächsten 10 Züge aus `aiSolution`.
  5. Ruft `drawPathInGreen()` auf, um diese Züge zu visualisieren.
  6. Benachrichtigt den Spieler über die angezeigten Schritte.

### 5.3. Labyrinthregeneration

`regenerateMaze()`

- **Zweck:** Regeneriert das Labyrinth in festgelegten Intervallen und versucht, die Spielerpositionen zu bewahren.
- **Prozess:**
  1. Überprüft, ob bereits eine Regeneration im Gange ist oder ob das Spiel läuft.
  2. Speichert die aktuellen Positionen der Spieler (und der KI, wenn im Wettkampfmodus) mithilfe von `Point`-Instanzen.
  3. Speichert die aktuelle Wandkonfiguration, um sie bei Bedarf wiederherstellen zu können.
  4. Ruft `generate()` auf, um ein neues Labyrinth zu erstellen.
  5. Stellt sicher, dass Pfade von den gespeicherten Positionen zum Ausgang existieren, indem `ensurePathExists(Point start)` aufgerufen wird.
  6. Wenn kein gültiger Pfad existiert, stellt die alte Labyrinthkonfiguration wieder her.
  7. Positioniert die Spieler an ihren gespeicherten Standorten neu.
  8. Zeichnet das Labyrinth und die Spieler neu.
  9. Setzt das Regenerationsflag zurück und benachrichtigt den Spieler.

### 5.4. Spiel speichern und laden

`saveGame()`

- **Zweck:** Speichert den aktuellen Spielstand in einer Datei (`savegame.dat`) unter Verwendung von Java-Serialisierung.
- **Prozess:**

1. Öffnet einen `ObjectOutputStream` zu `savegame.dat`.
2. Schreibt wesentliche Spieldaten, einschließlich Labyrinthabmessungen, Gitter, Spielerpositionen, Zugzähler und verstrichene Zeit.
3. Schließt den Stream und bestätigt die Speicherung gegenüber dem Spieler.

`loadGame()`

- **Zweck:** Lädt einen gespeicherten Spielstand aus `savegame.dat` unter Verwendung von Java-Deserialisierung.
- **Prozess:**
  1. Öffnet einen `ObjectInputStream` von `savegame.dat`.
  2. Liest und stellt die Spieldaten wieder her, einschließlich Labyrinthabmessungen, Gitter, Spielerpositionen, Zugzähler und verstrichene Zeit.
  3. Wenn im Wettkampfmodus, berechnet die KI ihren Lösungspfad neu.
  4. Setzt das Spiel als laufend und zeichnet das Labyrinth und die Spieler neu.
  5. Bestätigt das Laden gegenüber dem Spieler.

## 5.5. Spielverwaltung

`startGame()`

- **Zweck:** Startet ein Einzelspielerspiel.
- **Prozess:**
  1. Setzt den menschlichen Zugzähler zurück.
  2. Ruft `generate()` auf, um ein neues Labyrinth zu erstellen.
  3. Ruft `draw()` auf, um das Labyrinth zu rendern.
  4. Initialisiert den Spieler am Eingang durch Aufruf von `initializePlayer()`.
  5. Startet den Spieltimer mit `startTimer()`.
  6. Startet den Labyrinthregenerationstimer mit `startRegenerationTimer()`.
  7. Betritt die Hauptspielfunktion durch Aufruf von `runGameLoop()`.

`startCompetitiveMode()`

- **Zweck:** Startet ein Wettkampfspiel, bei dem der Spieler gegen eine KI antritt, um zuerst den Ausgang zu erreichen.
- **Prozess:**
  1. Setzt den menschlichen Zugzähler zurück.
  2. Ruft `generate()` auf, um ein neues Labyrinth zu erstellen.
  3. Setzt `isCompetitiveMode` auf `true` und `isHumanTurn` auf `true`.
  4. Initialisiert sowohl den menschlichen Spieler als auch den KI-Spieler am Eingang.
  5. Löscht vorhandene KI-Lösungen und berechnet den Lösungspfad der KI neu mit `findSolution()`.
  6. Kopiert die KI-Lösung zu `aiPlannedMoves`.
  7. Ruft `draw()` auf, um das Labyrinth zu rendern.
  8. Zeichnet das Labyrinth und die Spieler neu mit `redrawMazeAndPlayer()`.
  9. Startet den Spieltimer mit `startTimer()`.
  10. Startet den Labyrinthregenerationstimer mit `startRegenerationTimer()`.
  11. Betritt die Wettkampfspielfunktion durch Aufruf von `runCompetitiveGameLoop()`.

## 5.6. Spielinitialisierung

**play()**

- **Zweck:** Dient als Haupteinstiegspunkt für das Spiel und behandelt Benutzerinteraktionen zum Starten oder Laden von Spielen.
- **Prozess:**

1. Zeigt eine Willkommensnachricht und Menüoptionen an:

- 1) Letztes gespeichertes Spiel laden
- 2) Neues Spiel starten

2. Fordert den Benutzer auf, seine Wahl (1 oder 2) einzugeben und validiert die Eingabe.

3. Wenn der Benutzer ein Spiel laden möchte (1):

- Ruft `loadGame()` auf.
- Wenn das Laden erfolgreich ist und `isGameRunning` auf `true` gesetzt ist, ruft `continueGame()` auf, um fortzufahren.
- Wenn das Laden fehlschlägt, benachrichtigt den Benutzer und beginnt ein neues Spiel.

4. Wenn der Benutzer ein neues Spiel starten möchte (2):

- Fordert den Benutzer auf, einen Schwierigkeitsgrad auszuwählen:
  - 1) Einfach (10x10)
  - 2) Mittel (20x20)
  - 3) Schwer (30x30)
- Liest und validiert die Schwierigkeitswahl und setzt die Labyrinthgröße entsprechend (10, 20 oder 30).
- Fordert den Benutzer auf, einen Spielmodus auszuwählen:
  - 1) Einzelspieler
  - 2) Wettkampf (gegen KI)
- Liest und validiert die Spielmoduswahl.
- Initialisiert das Labyrinthgitter mit den gewählten Einstellungen durch Aufruf von `initializeGrid()`.
- Initialisiert den Spieler durch Aufruf von `initializePlayer()`, wenn im Einzelspielermodus.
- Startet das Spiel durch Aufruf von `startGame()` oder `startCompetitiveMode()` basierend auf dem gewählten Modus.

## 5.7. Verwaltung der Spielschleife

**runGameLoop()**

- **Zweck:** Verwaltet die Hauptschleife für den Einzelspielermodus, behandelt Benutzereingaben und den Spielfortschritt.
- **Prozess:**

1. Läuft kontinuierlich, solange `isGameRunning` auf `true` gesetzt ist.

2. Fordert den Spieler zur Eingabe von Befehlen auf:

- Bewegungsbefehle: W, A, S, D
- Andere Befehle: solve, next, save, load, q

3. Liest und verarbeitet die Eingabe:

- **q**: Beendet das Spiel durch Aufruf von `stopGame()`.
- **solve**: Ruft `solveMaze()` auf, um den vollständigen Lösungspfad anzuzeigen.
- **next**: Ruft `provideNext10Steps()` auf, um die nächsten 10 Schritte der Lösung anzuzeigen.
- **save**: Ruft `saveGame()` auf, um den aktuellen Spielstand zu speichern.
- **load**: Ruft `loadGame()` auf, um einen gespeicherten Spielstand zu laden.
- **Bewegungsbefehle (W, A, S, D)**: Ruft `movePlayer(char direction)` mit der entsprechenden Richtung auf.

4. Setzt die Schleife fort, bis das Spiel gestoppt oder der Spieler beendet.

### 5.8. KI-Pfadausführung

`makeAIMove()`

- **Zweck**: Führt den nächsten Zug der KI basierend auf ihrem vorgeplanten Lösungspfad aus.
- **Prozess**:

1. Überprüft, ob `aiPlannedMoves` nicht leer ist.
2. Ruft den nächsten Zug (`char`) aus `aiPlannedMoves` ab.
3. Ruft `moveAIPlayer(char direction)` auf, um den Zug der KI auszuführen.
4. Erhöht den KI-Zugzähler (`aiMoveCount`).
5. Überprüft, ob die KI den Ausgang erreicht hat, indem `checkAIWinCondition()` aufgerufen wird.
6. Wechselt den Zug zurück zum menschlichen Spieler, indem `isHumanTurn` auf `true` gesetzt wird.
7. Zeigt die aktuellen Zugzähler für beide Spieler an.

## III. Zusammenfassung der Methoden

Funktionalität	Hauptmethoden	Hilfsmethoden
<b>Labyrinthgenerierung</b>	<code>generate()</code> , <code>regenerateMaze()</code>	<code>resetGrid()</code> , <code>getUnvisitedNeighbor()</code> , <code>removeWalls()</code> , <code>createEntranceAndExit()</code> , <code>createOpeningOnSide()</code>
<b>KI-Lösung</b>	<code>findSolution()</code> , <code>solveMaze()</code> , <code>provideNext10Steps()</code>	<code>getDeltaFromDirection()</code> , <code>getCharFromDirection()</code>
<b>Spielverwaltung</b>	<code>startGame()</code> , <code>startCompetitiveMode()</code>	<code>runGameLoop()</code> , <code>runCompetitiveGameLoop()</code> , <code>makeAIMove()</code>
<b>Spielinitialisierung</b>	<code>play()</code>	<code>initializeGrid()</code> , <code>initializePlayer()</code>
<b>Verwaltung der Spielschleife</b>	<code>runGameLoop()</code>	<i>Keine</i>

Funktionalität	Hauptmethoden	Hilfsmethoden
<b>KI-Pfadausführung</b>	makeAIMove()	moveAIPlayer(), updateAIPosition(), checkAIWinCondition()
<b>Spiel speichern/laden</b>	saveGame(), loadGame()	<i>Keine</i>
<b>Rendering</b>	draw(), redrawMazeAndPlayer(), drawPathInGreen()	setupTurtle(), drawMazeStructure(), drawOuterBorder(), drawCellWalls(), drawWall(), drawPlayerPosition(), drawBothPlayers(), colorEntranceAndExit(), drawCellOpening(), drawTimer(), formatTime()
<b>Bewegungsverarbeitung</b>	movePlayer()	getMovementDeltas(), canMove(), getDirectionIndex(), updatePlayerPosition(), checkWinCondition()
<b>Timerverwaltung</b>	startTimer(), stopGame()	<i>Keine</i>
<b>Pfadzeichnung</b>	drawPathInGreen()	<i>Keine</i>
<b>Hilfsmethoden</b>	<i>Keine</i>	isValidCell(), getCell(), delay(), initializeGrid(), initializePlayer()

Hier ist die korrigierte Version Ihres Textes mit ordentlicher Formatierung, Grammatik und Struktur. Ich habe alle Kommentare (/\*\*/) wie gewünscht beibehalten:

---

### III. Beispiele für Spielszenarien

#### Spielinitialisierung:

##### Erstellen der Turtle:

###### Terminal

```
jshell> Turtle turtle = new Turtle(800, 800)
turtle ==> Turtle@70a9f84e
```

##### Erstellen einer Maze-Instanz:

###### Terminal

```
jshell> Maze maze = new Maze(turtle, 10, 10, 10) /* Erstellen eines Labyrinths
maze ==> Maze@3444d69d
```

## Scenario 1 & 2: Single-Player Mode With AI Help

### 1. Starting the Game:

- The player launches the Maze Game and selects **Start New Game**.

#### Terminal

```
jshell> maze.play();
Welcome to the Maze Game!
1) Load Last Saved Game
2) Start New Game
Enter your choice (1 or 2): // the player chooses 2
```

- Wählt den Schwierigkeitsgrad **Easy (10x10)**.

#### Terminal

```
Select difficulty:
1) Easy (10x10)           // the player chooses 1
2) Medium (20x20)
3) Hard (30x30)
```

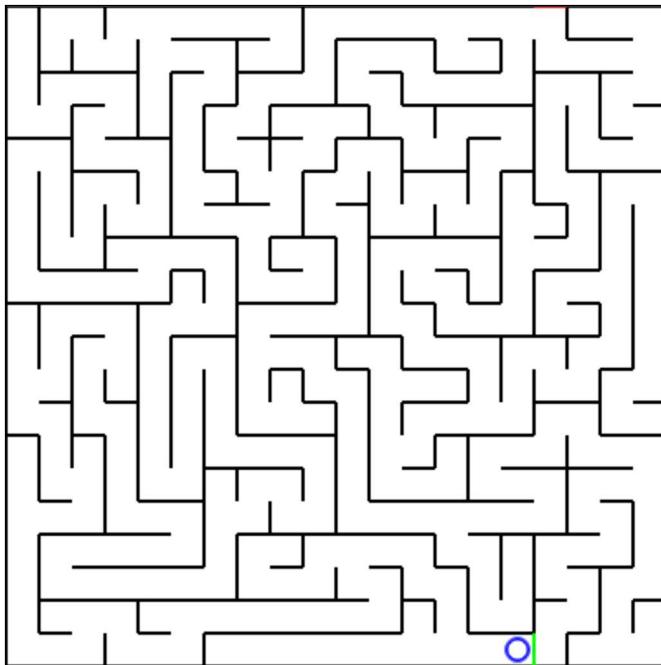
- Wählt **Single Player** als Spielmodus.

#### Terminal

```
Select game mode:
1) Single Player          // the player chooses 1
2) Competitive (vs AI)
```

### 2. Navigieren im Labyrinth:

- Das Labyrinth wird generiert und angezeigt.



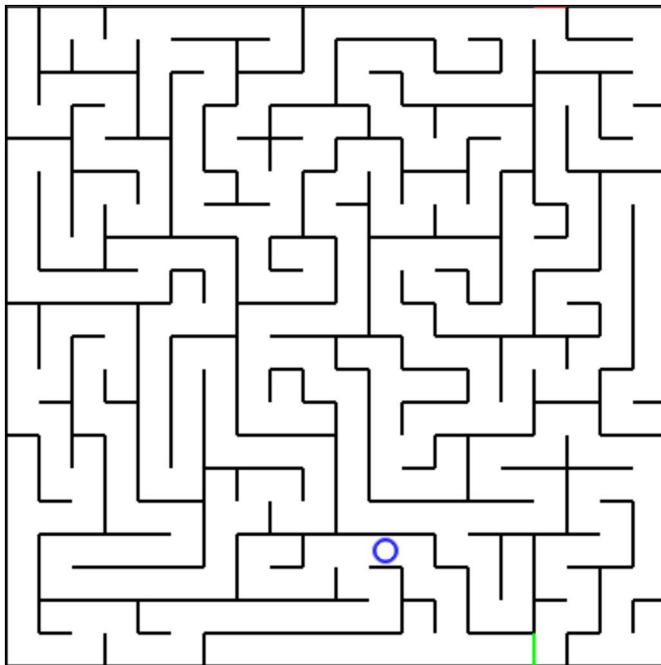
00:00

Moves: 0

- Der Spieler verwendet die Tasten W, A, S, D, um nach oben, links, unten oder rechts zu bewegen.

**Terminal**

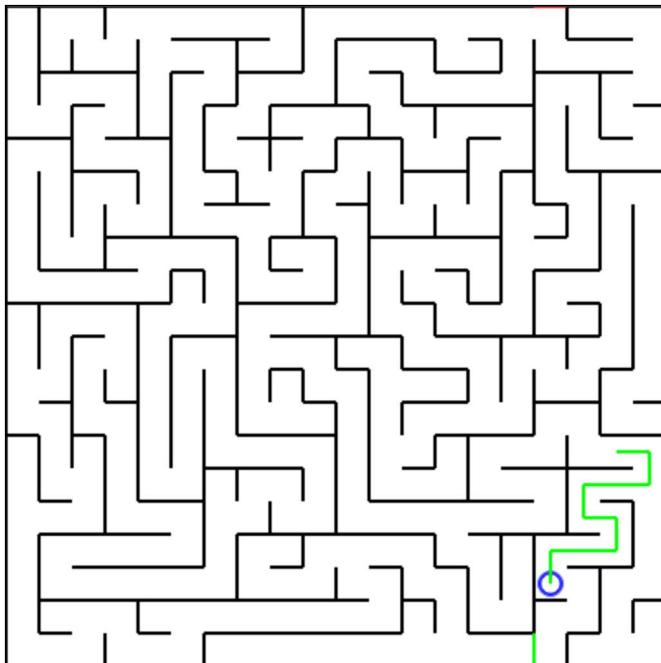
```
Move (WASD/solve/next/save/load/q) :
```



00:09

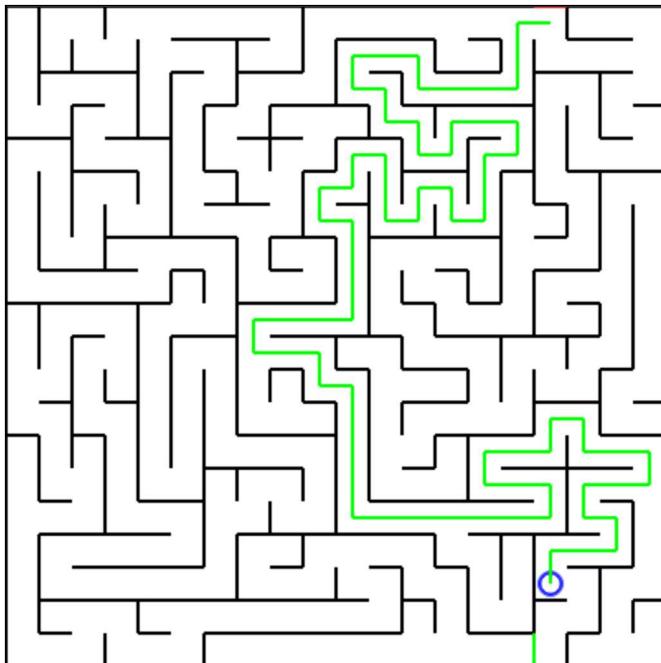
Moves: 6

- Wenn der Spieler feststeckt, gibt er `solve` ein, um den vollständigen Lösungspfad anzuzeigen, oder `next`, um die nächsten 10 Schritte zu sehen.



00:31

Moves: 19



00:40

Moves: 19

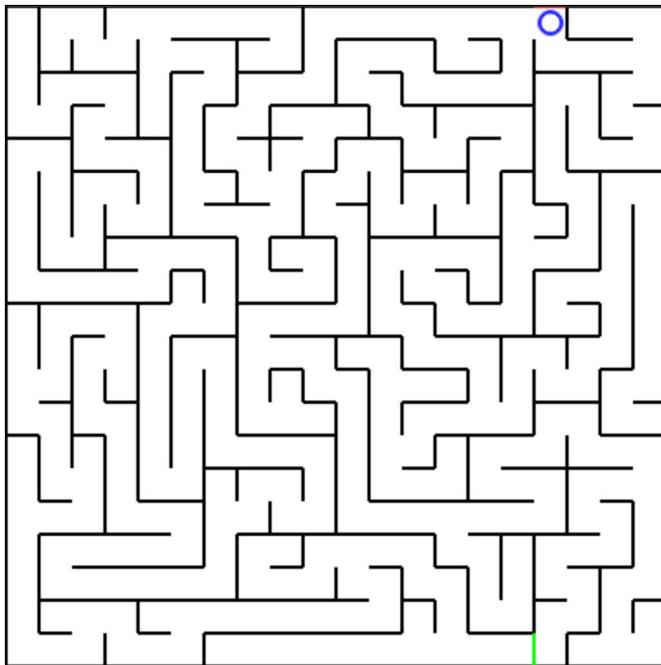
### 3. Spielgewinn:

- Beim Erreichen des Ausgangs gratuliert das Spiel dem Spieler.

#### Terminal

Congratulations! You completed the maze in 53 seconds.

- Zeigt die insgesamt benötigte Zeit und die Anzahl der gemachten Züge an.



01:42

Moves: 93

### Szenario 3: Competitive Mode

#### 1. Spiel starten:

- Der Spieler startet das Labyrinthspiel und wählt **Competitive Mode**.
- Der Spieler wählt **Start New Game**.

#### Terminal

```
jshell> maze.play();
Welcome to the Maze Game!
1) Load Last Saved Game
2) Start New Game
Enter your choice (1 or 2): // the player chooses 2
```

- Wählt den Schwierigkeitsgrad **Medium (20x20)**.

#### Terminal

Select difficulty:

- 1) Easy (10x10)
- 2) Medium (20x20) // the player chooses 2
- 3) Hard (30x30)

- Wählt **Competitive (vs AI)** als Spielmodus.

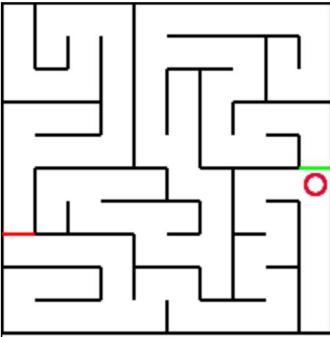
#### Terminal

Select game mode:

- 1) Single Player
- 2) Competitive (vs AI) // the player chooses 2

#### 2. Spielverlauf:

- Sowohl der Spieler als auch die KI starten am Eingang des Labyrinths.

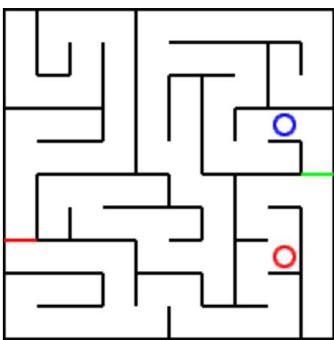


00:04

Moves: 0/0

#### Terminal

Competitive Mode! Take turns moving.  
Use WASD keys to move, 'q' to quit.  
Your turn (WASD/save/load/q): w  
Moves - You: 1, AI: 0  
AI's turn...  
Moves - You: 1, AI: 1  
Your turn (WASD/save/load/q): d  
Moves - You: 2, AI: 1  
AI's turn...  
Moves - You: 2, AI: 2  
Your turn (WASD/save/load/q): s  
Moves - You: 3, AI: 2  
AI's turn...  
Moves - You: 3, AI: 3  
Your turn (WASD/save/load/q): d  
Moves - You: 4, AI: 3  
...



00:16

Moves: 5/5

### 3. Winning the Game:

- Der Erste, der den Ausgang erreicht, gewinnt.
- Das Spiel kündigt den Gewinner an und liefert Statistiken zu Zügen und verstrichener Zeit.

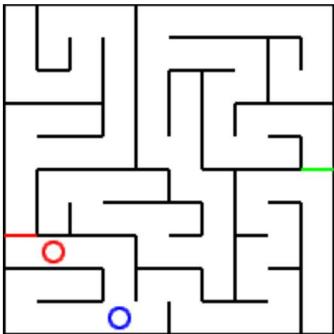
**Terminal**

Game Over! AI wins in 50 moves!

Time elapsed: 25 seconds

Your moves: 50

Moves - You: 50, AI: 50



00:38

Moves: 28/28

**Szenario 4: Labyrinthregeneration****1. Spiel starten:**

- Der Spieler startet das Labyrinthspiel und wählt **Start New Game**.

**Terminal**

```
jshell> maze.play();
Welcome to the Maze Game!
1) Load Last Saved Game
2) Start New Game
Enter your choice (1 or 2): // the player chooses 2
```

- Wählt den Schwierigkeitsgrad **Hard (30x30)**.

**Terminal**

```
Select difficulty:
1) Easy (10x10)
2) Medium (20x20)
3) Hard (30x30) // the player chooses 1
```

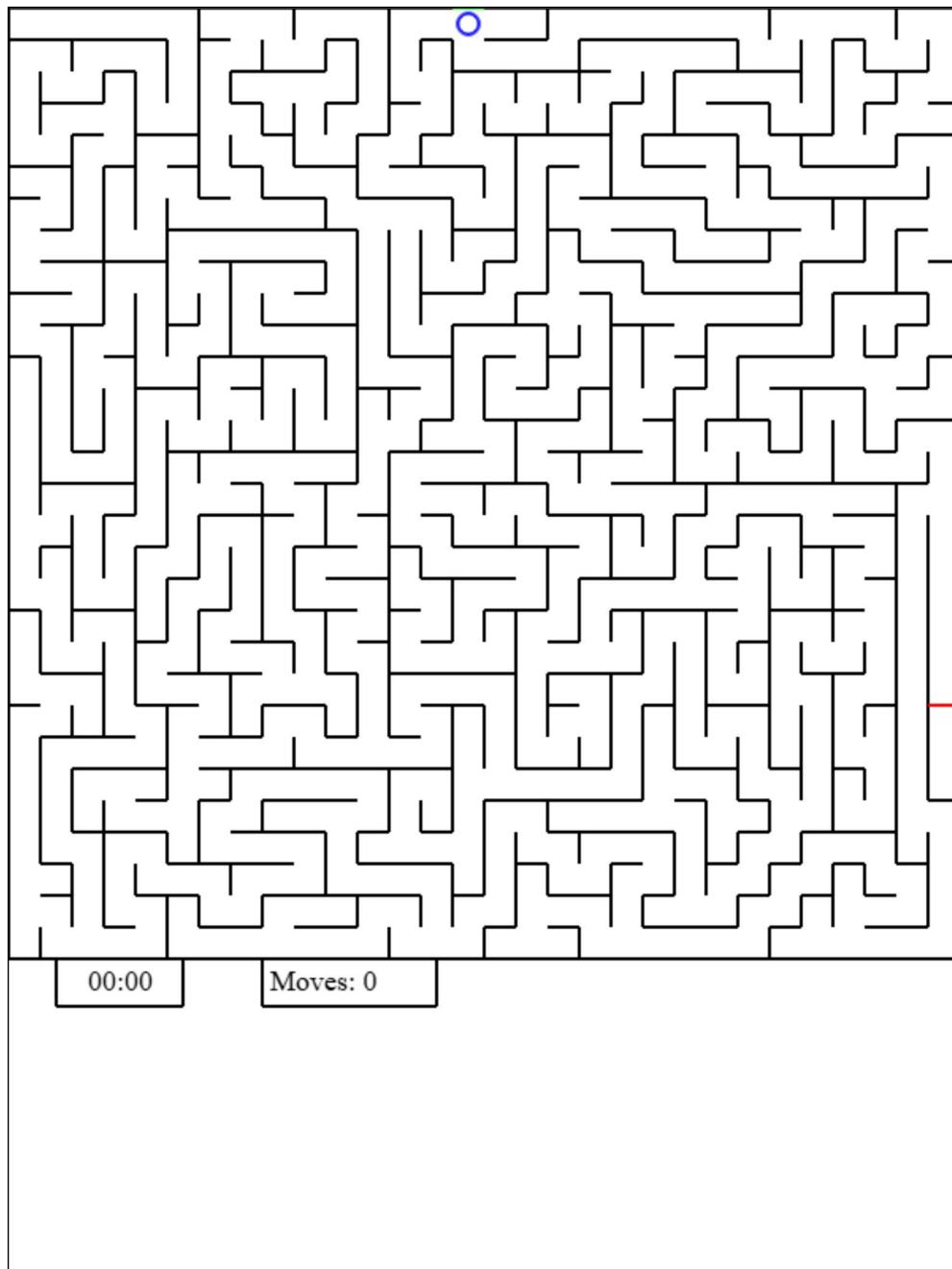
- Wählt **Single Player** als Spielmodus.

**Terminal**

```
Select game mode:
1) Single Player // the player chooses 1
2) Competitive (vs AI)
```

**2. Navigieren im Labyrinth:**

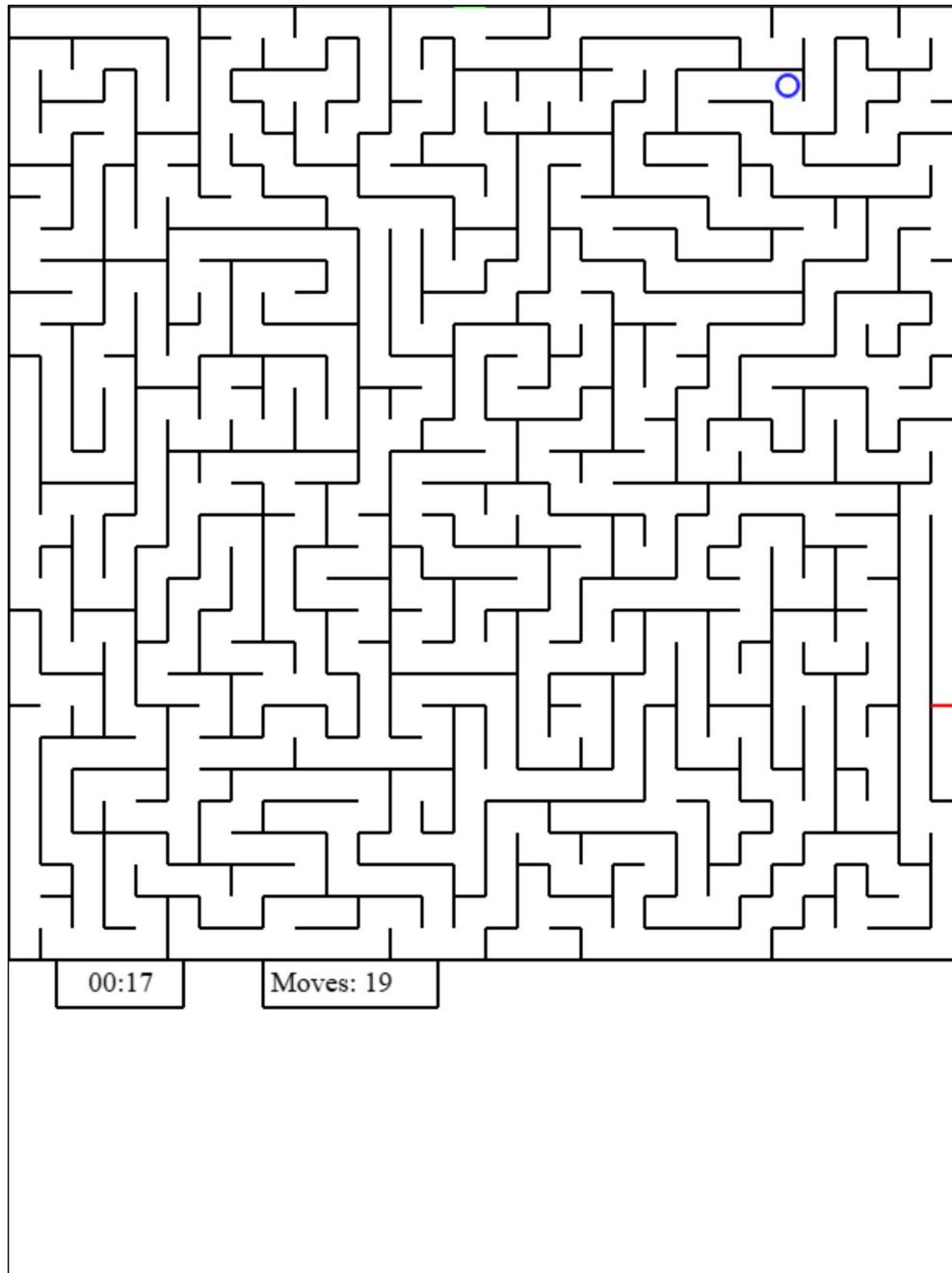
- Das Labyrinth wird generiert und angezeigt.



- Der Spieler verwendet die Tasten W, A, S, D, um nach oben, links, unten oder rechts zu bewegen.

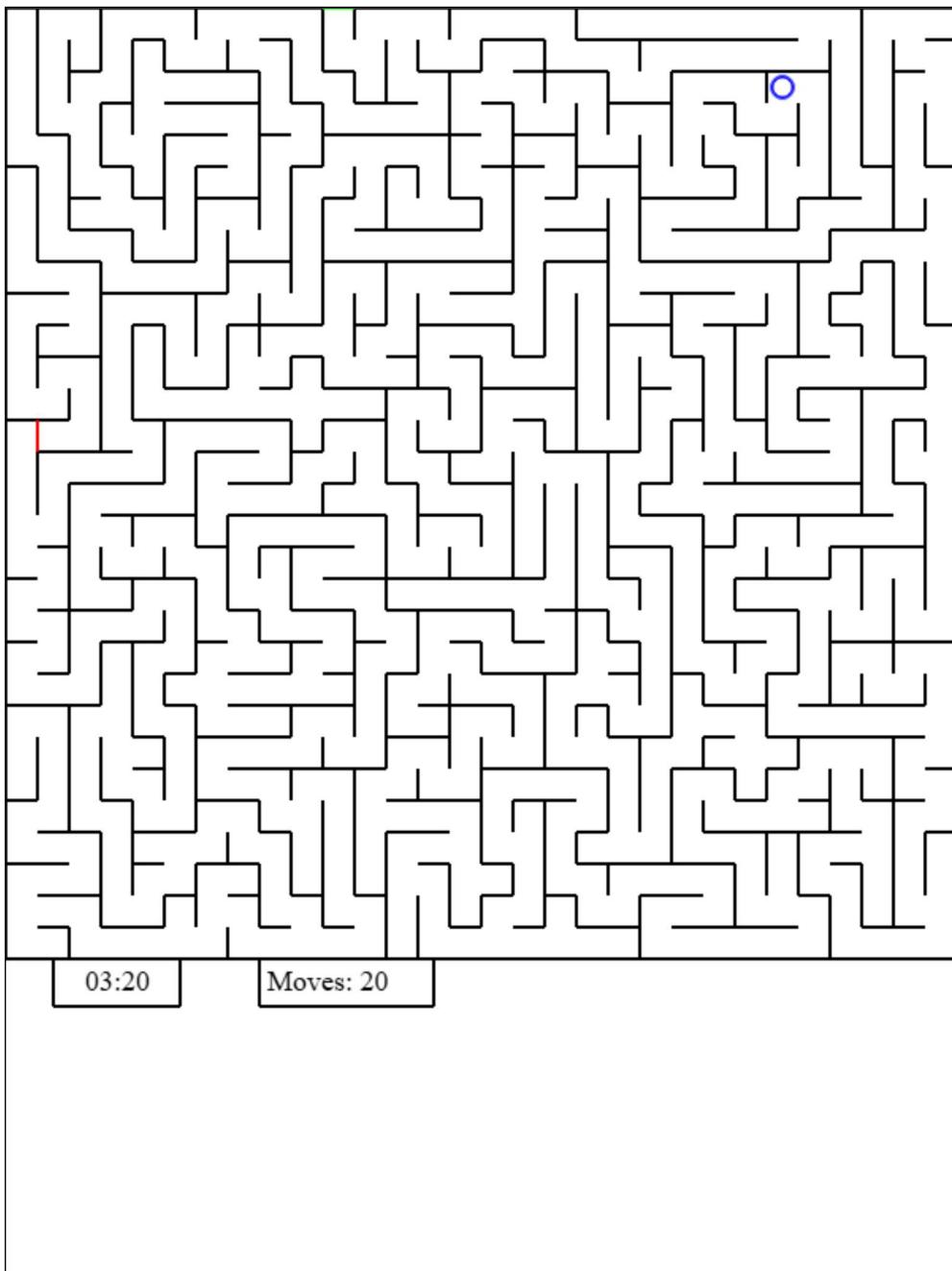
**Terminal**

Move (WASD/solve/next/save/load/q) :



### 3. Labyrinthregeneration:

- Wenn der Spieler das Spiel nicht innerhalb von 180 Sekunden beendet, regeneriert sich das Labyrinth und der Spieler bleibt an derselben Position.



### Terminal

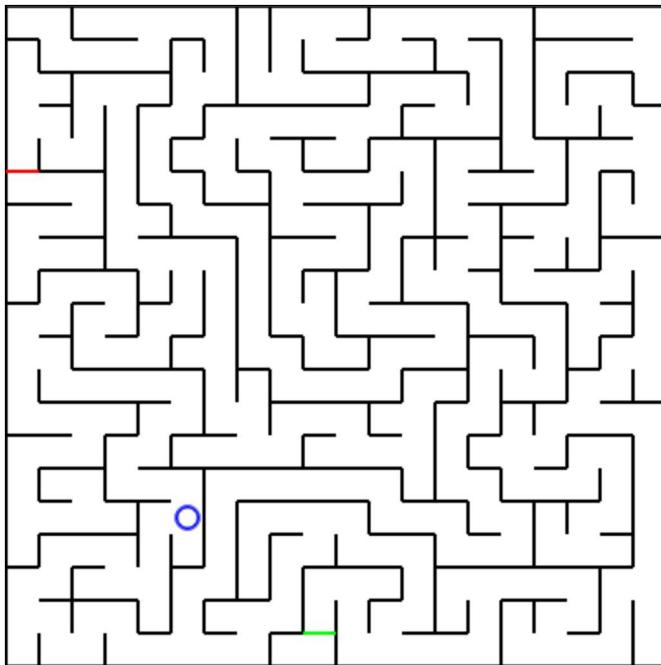
```
Maze regenerated! Keep going!
Move (WASD/solve/next/save/load/q) :
```

- Der Spieler kann weiterhin die Methoden `next` oder `solve` verwenden.
- **Hinweis:** Das Labyrinth regeneriert sich auch im Wettkampfmodus.

### Szenario 5: Spiel speichern und laden

#### 1. Spiel speichern:

- Der Spieler startet ein neues Spiel.
- Während des Spiels entscheidet sich der Spieler, den Fortschritt zu speichern.



00:27

Moves: 39

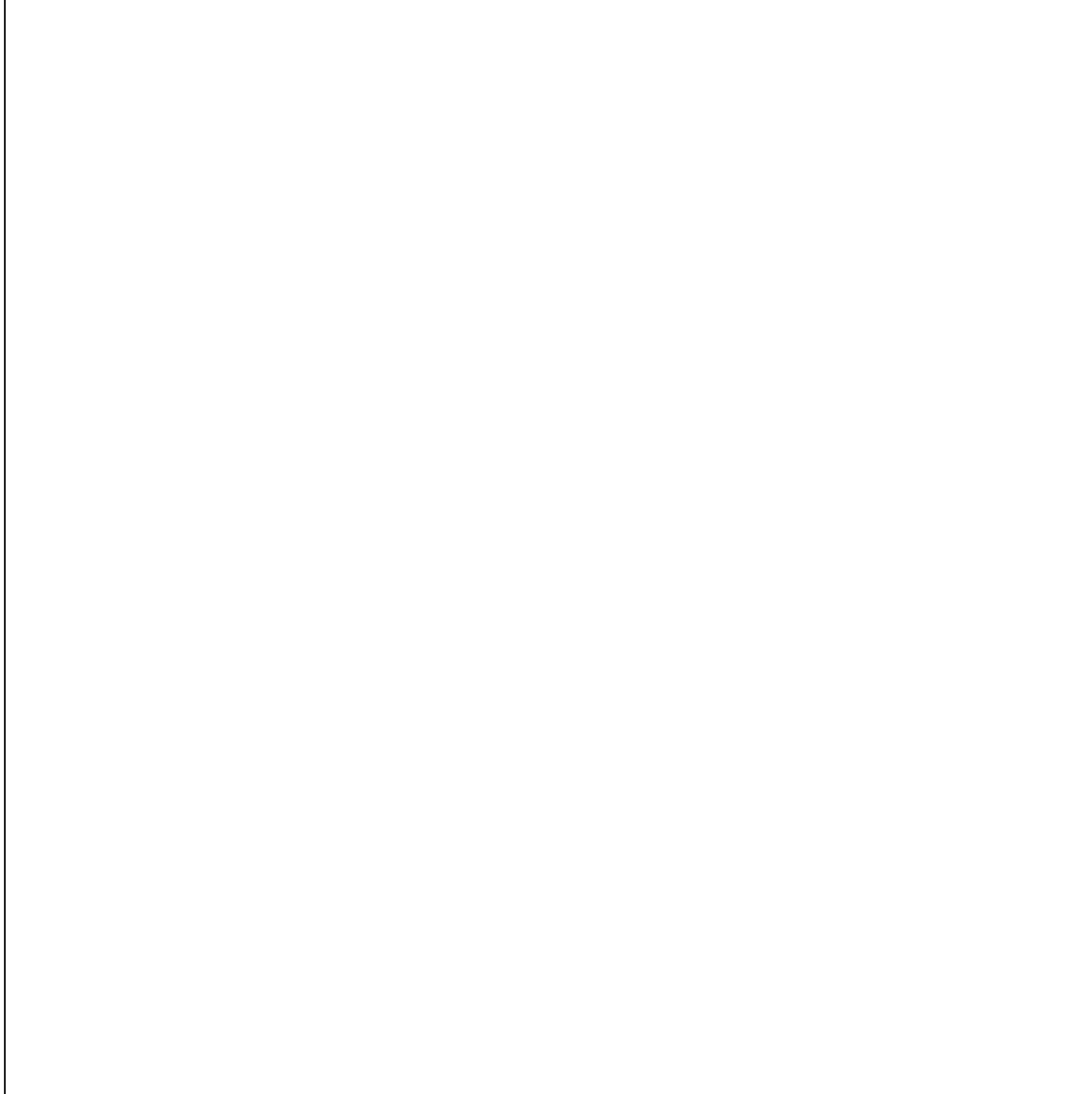
- Gibt `save` ein und der Spielstand wird in `savegame.dat` geschrieben.

**Terminal**

```
Move (WASD/solve/next/save/load/q): save
Game saved successfully.
```

**2. Spiel laden:**

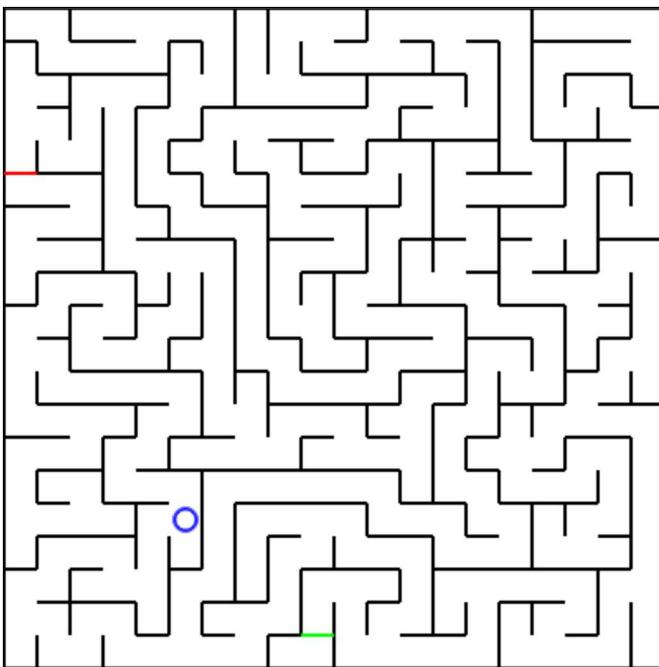
- Später setzt der Spieler fort, indem er **Letztes gespeichertes Spiel laden** auswählt.



### Terminal

```
jshell> maze.play()
Welcome to the Maze Game!
1) Load Last Saved Game
2) Start New Game
Enter your choice (1 or 2): 1
Game loaded successfully.
Continuing the loaded game...
Move (WASD/solve/next/save/load/q) :
```

- Das Spiel stellt das Labyrinth, die Spielerpositionen, die Zugzähler und die verstrichene Zeit aus `savegame.dat` wieder her.



00:28

Moves: 40

---

### 3. Fortsetzung des Spiels:

- Der Spieler setzt die Navigation im Labyrinth von dort fort, wo er aufgehört hat.
- 

## 11. Fazit

Diese umfassende Dokumentation bietet einen strukturierten und detaillierten Überblick über die Architektur, Funktionen und den Betriebsablauf des Labyrinthspiels. Sie umfasst alle Haupt- und Hilfsmethoden innerhalb der `Maze`-Klasse sowie die unterstützenden Klassen `Cell`, `Player` und `Point`.