

Dokumentation Bachelorprojekt

Simulator zur Evaluation von Explorationsstrategien für aktive modellfreie
Reinforcement Learning Verfahren in dynamischen Umgebungen

Philip Martin

Matrikelnummer: /

Timo Sturm

Matrikelnummer: /

Kassel, den 10. Juni 2020

Inhaltsverzeichnis

1	Einleitung	4
2	Allgemein	6
2.1	Dynamisches Maze-Problem	6
2.2	Simulationsablauf	8
2.3	Reinforcement Learning	9
3	Benutzerschnittstellen	10
3.1	Starten der Anwendung	10
3.2	Simulator-UI	11
3.2.1	Simulationsansicht	11
3.2.2	Infoansicht	13
3.2.3	UI-Log	16
3.3	Konfigurationsassistent	17
3.3.1	Konfigurationsansicht	18
3.3.2	Erweiterbarkeit und Aufbau	21
4	Konfiguration	22
4.1	Erstellen einer Konfiguration	22
4.2	Erweiterung der Konfiguration	23
4.2.1	Einfügen neuer Werte	23
4.2.2	Einfügen neuer Schlüssel	26
5	Labyrinth	28
5.1	Initiales Level	28
5.1.1	Aufbau	28
5.1.2	Konfiguration	29

5.2	Levelveränderung	31
5.2.1	Algorithmus	31
5.2.2	Operatoren	32
5.3	Komplexität	36
5.3.1	DefaultComplexityFunction	37
6	Agent	41
6.1	Lernverfahren	41
6.1.1	Kodierung der Zustände	41
6.1.2	Q-Learning	43
6.2	Explorationsstrategien	44
6.2.1	Greedy-Strategie	44
6.2.2	Random-Strategie	45
6.2.3	Epsilon-Greedy-Strategie	45
6.2.4	Decreasing-Epsilon-Strategie	45
6.2.5	Epsilon-First-Strategie	46
6.2.6	Softmax-Strategie	47
6.2.7	VDBE-Strategie	47
6.2.8	Hinzufügen einer Explorationsstrategie	48
7	Bedingungen	50
7.1	Episodenabbruch	50
7.1.1	Endzustand erreicht	50
7.1.2	Maximale Schrittzahl erreicht	51
7.1.3	Optimale Pfadlänge überschritten	51
7.2	Levelveränderung	52
7.2.1	Maximale Episodenanzahl erreicht	52
7.2.2	Bestimmte Performance erreicht	52
7.3	Hinzufügen einer Bedingung	54
8	Logger	55
8.1	Speicherort	55
8.2	Inhalt	55
8.2.1	miscLog.txt	57
8.2.2	qtable.csv	58

8.2.3	evaluation.csv	59
8.2.4	maze.png	59
8.2.5	<trainingName>.cfg	60
8.2.6	summary.csv	60
8.3	Funktionsweise	61
8.3.1	Aufbau	62
8.3.2	Erweiterung	63
9	Schluss	65
9.1	Probleme	65
9.2	Ausblick	67
	Übersicht Konfiguration	71
	Übersicht Zuständigkeit	72

1 Einleitung

Diese Ausarbeitung stellt die Dokumentation des Bachelorprojektes von Philip Martin und Timo Sturm dar, welches im Laufe des Wintersemesters 19/20 an der Universität Kassel im Fachgebiet Intelligente Eingebettete Systeme absolviert wurde. Ziel des Projektes war die Entwicklung eines Simulators, welche die Evaluierung von Explorationsstrategien für aktive modellfreie Reinforcement Learning (RL) Verfahren in dynamischen Umgebungen ermöglicht. Die Simulationsumgebung sollte hierbei auf dem aus der RL-Welt bekannten Maze-Problem basieren, welches sich im Laufe des Trainings dynamisch verändert. Hierfür sollte das Training in verschiedene Level, welche selbst wiederum in Episoden organisiert sein sollten, unterteilt werden.

Die Dokumentation beginnt in Kapitel 2 mit einer detaillierten Beschreibung des klassischen Maze-Problems und des auf ihm aufbauenden dynamischen Maze-Problems, welches die Grundlage der Simulationsumgebung darstellt. An dieser Stelle wird zudem der grundlegende Ablauf der Simulation skizziert. Aufbauend auf diesen Informationen wird am Ende des Kapitels ein Bezug zwischen dem (dynamischen) Maze-Problem und dem Thema Reinforcement Learning hergestellt. Anschließend wird in Kapitel 3 beschrieben, wie der Benutzer die Simulation starten kann und welche Freiheiten er hierbei hat. Im Rahmen des Projektes wurden zwei (optionale) graphische Oberflächen entwickelt, dessen Funktionalitäten in diesem Kapitel genauer erläutert werden. Eine dieser Oberfläche visualisiert das Training und präsentiert dem Benutzer in Echtzeit einige der gesammelten Daten, wohingegen die andere Oberfläche einen Konfigurationsassistenten darstellt, welcher eine Alternative zur direkten Bearbeitung der Konfigurationsdatei bietet. Anschließend wird in Kapitel 4 beschrieben, wie der Benutzer den Simulator konfigurieren kann. Hierbei wird zudem skizziert werden, wie der Benutzer die Konfigurationsmöglichkeiten erweitern kann, indem er einen bereits bestehenden Schlüssel um neue mögliche Werte erweitern oder ggf. gänzlich neue Schlüssel einpflegen kann. Daraufhin werden in Kapitel 5 diverse Informationen hinsichtlich des verwendeten Labyrinths aufgeführt. Hierbei wird

erklärt, wie das initiale Labyrinth generiert wird und wie dieses im weiteren Verlauf der Simulation durch die Anwendung diverser Operatoren verkompliziert wird. Des Weiteren wird in diesem Kapitel erläutert, wie derzeit die Komplexität eines Labyrinthes berechnet wird, um so ggf. unterschiedliche Labyrinthe miteinander vergleichen zu können. Im Anschluss hieran wird in Kapitel 7 beschrieben, was für Bedingungen der Benutzer derzeit konfigurieren kann, um das Ende eines Levels oder einer Episode herbeizuführen. Im Anschluss wird in Kapitel 8 der Logger betrachtet. Hierbei wird erklärt, welche Informationen der Logger protokolliert und in welcher Form diese persistent gespeichert werden. Zudem wird kurz skizziert, wie die Klassenarchitektur des Loggers aufgebaut ist und wie das Logging ggf. um neue Informationen erweitert werden kann. Abschließend wird das Projekt in Kapitel 9 reflektiert, indem Aussagen über bestehende Probleme und mögliche Erweiterungen getroffen werden.

Es sei zudem darauf hingewiesen, dass sich im Anhang auf Seite 71 eine Tabelle finden lässt, welche alle Schlüssel der Konfiguration auflistet und jeweils auf den Abschnitt der Doku verweist, welcher sie beschreibt. Außerdem befindet sich auf Seite 72 eine weitere Tabelle, welche aufzeigt, welcher Entwickler für welche Funktionalität des Simulators verantwortlich ist.

Des Weiteren sei an dieser Stelle angemerkt, dass das komplette Projekt als Open-Source bereitgestellt wird und sich auf GitHub unter der folgenden Adresse finden lässt: <http://github.com/PhilipMar/Reinforcement-Learning-in-Dynamic-Environments-Simulator>

2 Allgemein

Wie bereits in der Einleitung erwähnt wurde, war das Ziel des Projektes die Entwicklung eines Simulators, welcher die Evaluierung von Explorationsstrategien für aktive modellfreie RL-Verfahren in dynamischen Umgebungen ermöglicht. Die Simulationsumgebung sollte hierbei auf dem aus der RL-Welt bekannten Maze-Problem basieren, welches sich im Laufe des Trainings dynamisch verändert. Dieses Kapitel beginnt daher mit einer Erklärung des klassischen und des auf ihm aufbauenden dynamischen Maze-Problems. Im Anschluss wird der grundlegende Ablauf der Simulation skizziert. Zudem wird abschließend der Bezug zwischen aktiven RL-Verfahren und dem zuvor eingeführtem Problem erläutert.

2.1 Dynamisches Maze-Problem

Das klassische Maze-Problem lässt sich in Kürze wie folgt beschreiben: Ein Agent muss in einem zweidimensionalen Labyrinth den kürzesten Weg von einem Start- zu einem Endfeld finden. Sowohl das Labyrinth als auch das Start- und Endfeld werden hierbei vorab vom Designer bestimmt. Des Weiteren lassen sich die Felder in passierbare und unpassierbare Felder unterteilen. So gibt es Felder, die der Agent betreten kann und Felder, die für den Agenten unzugänglich sind und demnach als Wände interpretiert werden können. Der Agent besitzt zu jedem Zeitpunkt nur Informationen über seine unmittelbare Umgebung. Demnach weiß er lediglich, welchem Typ (Weg, Wand) die Felder um ihn herum entsprechen. Der Agent kann sich durch das Labyrinth bewegen, indem er eine Aktion tätigt, wobei er sich mit einer Aktion nach oben, rechts, unten oder nach links bewegen kann, insofern diese Felder für ihn zugänglich sind.

Wie in der Einleitung dieses Kapitel erwähnt wurde, soll das beschriebene Ziel des Projektes durch die Bereitstellung einer dynamischen Simulationsumgebung erreicht wer-

den. Hierfür wird an dieser Stelle das „dynamische Maze-Problem“ eingeführt, welches das zuvor beschriebene klassische Maze-Problem um folgende Eigenschaften erweitert:

- Das Initiallabyrinth wird zufällig generiert.
- Ein Feld verfügt nun über eine Farbe, welche von einem Agenten in unmittelbarer Nähe wahrgenommen werden kann.
- Das Training besteht aus verschiedenen Levels.
- **Das Labyrinth verändert sich nach jedem Level und wird somit komplizierter.**
- Das Training ist erst vorbei, wenn alle Level beendet wurden.

Die wohl bedeutendste Neuerung des dynamischen Maze-Problems ist die levelweise erfolgende Veränderung des Labyrinths. Hierbei ist zu beachten, dass bei einer solchen Veränderung nicht ein gänzlich neues Labyrinth erzeugt wird, sondern lediglich Veränderungen/Erweiterungen am bestehenden Labyrinth vorgenommen werden. Hierfür wurden vier verschiedene Operatoren entwickelt, welche in Abschnitt 5.2.2 genauer vorgestellt werden. In Abbildung 2.1 wird exemplarisch die Entwicklung eines Mazes über vier Level hinweg dargestellt.

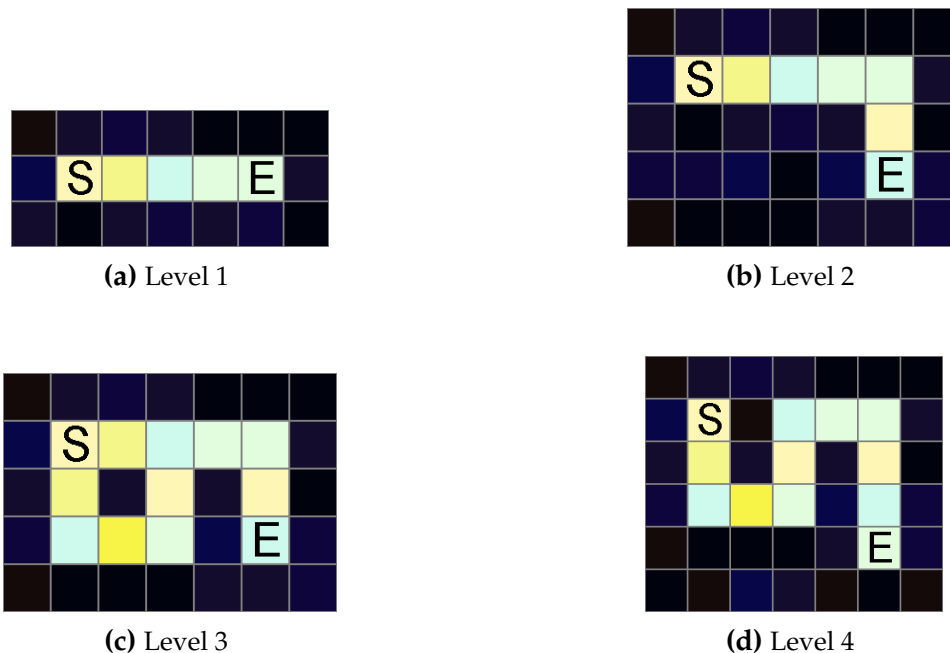


Abbildung 2.1: Beispiel eines dynamischen Maze-Problems

2.2 Simulationsablauf

In Algorithmus 1 wird der grundlegende Ablauf des dynamischen Maze-Problems skizziert, welcher im Detail in der Klasse `de.uni.k.s.Training` implementiert wurde. Zuerst müssen der Agent A und das initiale Labyrinth M initialisiert werden. Hiermit ist unter anderem die Generierung des initialen Labyrinthes gemeint, welche in Abschnitt 5.1 beschrieben wird. Anschließend muss die zuvor vom Benutzer bestimmte Levelanzahl *numberOfLevels* absolviert werden. Ein Level gilt hierbei als beendet, wenn eine der konfigurierten Levelveränderungsbedingungen (siehe Abschnitt 7.2) eingetreten ist, welche im Pseudocode mit *levelChangeCriteria* bezeichnet werden. Wie man zudem im Pseudocode erkennt, ist ein Level in Episoden unterteilt, innerhalb welcher der Agent seine Aktionen ausführt. Eine Bedingung für Levelveränderungen könnte daher beispielsweise eine festgelegte Anzahl an Episoden sein. Für Episoden gilt analog zu den Levels, dass eine der zuvor konfigurierten Episodenabbruchbedingungen (siehe Abschnitt 7.1) erfüllt werden muss, um es zu beenden. Die Bedingungen für Episodenabbrüche wurden im Pseudocode als *episodeStopCriteria* bezeichnet. Beispielsweise könnte die Erreichung des Endfeldes bzw. des Endzustandes eine solche Bedingung sein. Nachdem eine Levelveränderungsbedingung eingetreten ist, wird die Komplexität des Labyrinthes erhöht (siehe hierzu Abschnitt 5.3), indem das Labyrinth durch die Anwendung bestimmter zuvor konfigurierter Operatoren (siehe hierzu Abschnitt 5.2.2) verändert wird. Der Grad der Veränderung des Labyrinthes muss hierbei vom Benutzer bestimmt werden (siehe hierzu Abschnitt 5.2.1).

Algorithmus 1 Skizzierung des Ablaufes des dynamischen Maze-Problems

Require: *numberOfLevels*, *episodeStopCriteria*, *levelChangeCriteria*, ...

```
1: initialise agent  $A$ 
2: initialise maze  $M$ 
3: for levelNr  $\leftarrow 1$  to numberOfLevels do
4:   if levelNr  $\neq 1$  then
5:     change Maze  $M$ 
6:   while no criterion  $L \in \textit{levelChangeCriteria}$  is met do
7:     reset agent  $A$  for new episode
8:     while no criterion  $E \in \textit{episodeStopCriteria}$  is met do
9:       agent  $A$  perform action
```

2.3 Reinforcement Learning

Das Erlernen des kürzesten Weges zwischen dem Start- und dem Endfeld soll durch das Training mit einem RL-Verfahren realisiert werden. Hierfür müssen die dem Agent zur Verfügung stehenden Informationen in Zuständen kodiert werden, um ein für RL-Verfahren verarbeitbares Markow-Entscheidungsproblem zu erzeugen (siehe Abschnitt 6.1.1). Bei aktiven modellfreien RL-Verfahren führt der Agent in Abhängigkeit von der verwendeten Explorationsstrategie und seinem aktuellen Zustand Aktionen aus und speichert die „Ergebnisse“ dieser Aktionen in einer sogenannten Q-Table (siehe Abschnitt 6.1.2). So wird das Erreichen des Endzustandes i. d. R. mit einem positiven Reward honoriert. Allgemein gilt jedoch, dass jede Aktion, welche nicht das Erreichen des Endzustandes herbeiführt, dem Agenten einen negativen Reward einbringt. Die Explorationsstrategie bestimmt hierbei, ob der Agent das bisher gesammelte Wissen gierig ausnutzen soll (Exploitation) oder ob er eine Aktion ausführen soll, welche nach seinem aktuellem Wissensstand nicht die Erfolgsversprechendste ist, sein Wissen aber möglicherweise erweitern könnte (Exploration). Mithilfe des bereitgestellten Simulators soll der Benutzer die Möglichkeit haben, Explorationsstrategien hinsichtlich ihrer Tauglichkeit in dynamischen Umgebungen zu überprüfen. In Abschnitt 6.2 werden die bereits implementierte Explorationsstrategien kurz vorgestellt.

3 Benutzerschnittstellen

Im folgenden Kapitel werden die Benutzerschnittstellen des Simulators vorgestellt. Das Kapitel beginnt daher mit einer Anleitung, welche erklärt, wie die bereitgestellten Funktionalitäten über die Konsole aufgerufen werden können. Daraufhin werden die dabei aufrufbaren graphischen Oberflächen (Simulator-UI und Konfigurationsassistent) detailliert beschrieben. Hierbei wird sowohl auf den grundlegenden Aufbau der Oberflächen eingegangen als auch skizziert, wie sie ggf. erweitert werden können.

3.1 Starten der Anwendung

Der Simulator liegt als ausführbare `.jar` Datei vor, standardmäßig ist der volle Name `MazeSimulator.jar`. Ist auf dem Computer eine aktuelle Java-Version installiert, lässt sich das Programm über ein Terminal ausführen:

```
java -jar MazeSimulator.jar [argumente]
```

Dabei kann das Programm mit verschiedenen Argumenten gestartet werden, `--help` oder `-h` bietet im Terminal die Möglichkeit, alle Argumente mit einer kurzen Erklärung anzuzeigen. Um das Training mit einer Konfigurationsdatei (siehe das nachfolgende Kapitel 4 für Details) zu starten, wird dem Programm der Pfad zur entsprechenden Datei beim Start übergeben:

```
java -jar MazeSimulator.jar --config=path/to/myConfig.cfg
```

Natürlich muss die Konfigurationsdatei nicht in einem Ordner liegen, sondern kann direkt im selben Ordner wie die `.jar` liegen. Soll die graphische Benutzeroberfläche während des Trainings angezeigt werden, so muss beim Start noch zusätzlich das Argument `--showUI` übergeben werden. Um den Konfigurationsassistenten für die einfache Erstellung und Bearbeitung der Konfigurationsdatei zu starten, muss das Programm mit `--configure` als Argument gestartet werden. Wichtig hierbei ist, dass es nicht möglich ist, gleichzeitig das Training und den Konfigurationsassistenten zu starten. Hier noch einmal die möglichen Programmargumente in einer Übersicht:

Argument	Erklärung
<code>--config=</code>	Gibt den Pfad zur Konfigurationsdatei an
<code>--showUI</code>	Startet das Training mit der graphischen Benutzeroberfläche
<code>--configure</code>	Startet den Konfigurationsassistenten
<code>--help / -h</code>	Zeigt eine Übersicht über die möglichen Argumente

3.2 Simulator-UI

Wie bereits zuvor angedeutet wurde, besitzt der Benutzer die Möglichkeit, das Training mit einer graphischen Benutzeroberfläche zu starten. In den folgenden Unterabschnitten diese funktional beschrieben.

3.2.1 Simulationsansicht

In Abbildung 3.1 wird der Teil der UI hervorgehoben, der für die graphische Darstellung des Trainings zuständig ist.

Ganz oben im Bild befindet sich ein Label, welches anzeigt, in welchem Level und in welcher Episode der Agent sich befindet. Zudem wird hier angezeigt, wie viele Aktionen der Agent in der derzeitigen Episode bereits getätigt hat.

Weiter unten findet sich ein Canvas, welches das derzeitige Labyrinth des aktuellen Levels und die Position des Agenten in ihm visualisiert. Die Felder, auf welchen ein „S“ oder „E“ eingezeichnet ist, repräsentieren jeweils das Start- bzw. das Endfeld. Das „A“, welches sich

immer in einem grauen Kreis befindet, repräsentiert den Agenten und gibt somit seine aktuelle Position an.

Ganz unten im hervorgehobenen Teil des Bildes befindet sich ein Label und ein Slider. Mithilfe des Sliders lässt sich einstellen, wie lange (in Sekunden) die zeitliche Verzögerung zwischen zwei Aktionen sein soll. Wichtig ist an dieser Stelle, dass die hier eingestellte Verzögerung nicht zwingend der tatsächlichen Wartezeit zwischen zwei Aktionen entsprechen muss. Wird der Simulator mit der Benutzeroberfläche gestartet, werden die Logik der Benutzeroberfläche und das eigentliche Training in unterschiedlichen Threads ausgeführt. Es kann daher vorkommen, dass der Thread, der für die eigentliche Berechnung der Simulation zuständig ist, länger braucht, als es die eingestellte zeitliche Verzögerung eigentlich vorgeben würde. Zudem gilt, dass die mit dem Slider eingestellte Verzögerung immer nur an einem bestimmten Zeitpunkt übernommen bzw. aktualisiert wird. Es kann daher durchaus passieren, dass eine einmal eingestellte Verzögerung zwingend abgewartet werden muss, bevor die neue Wartezeit übernommen wird.

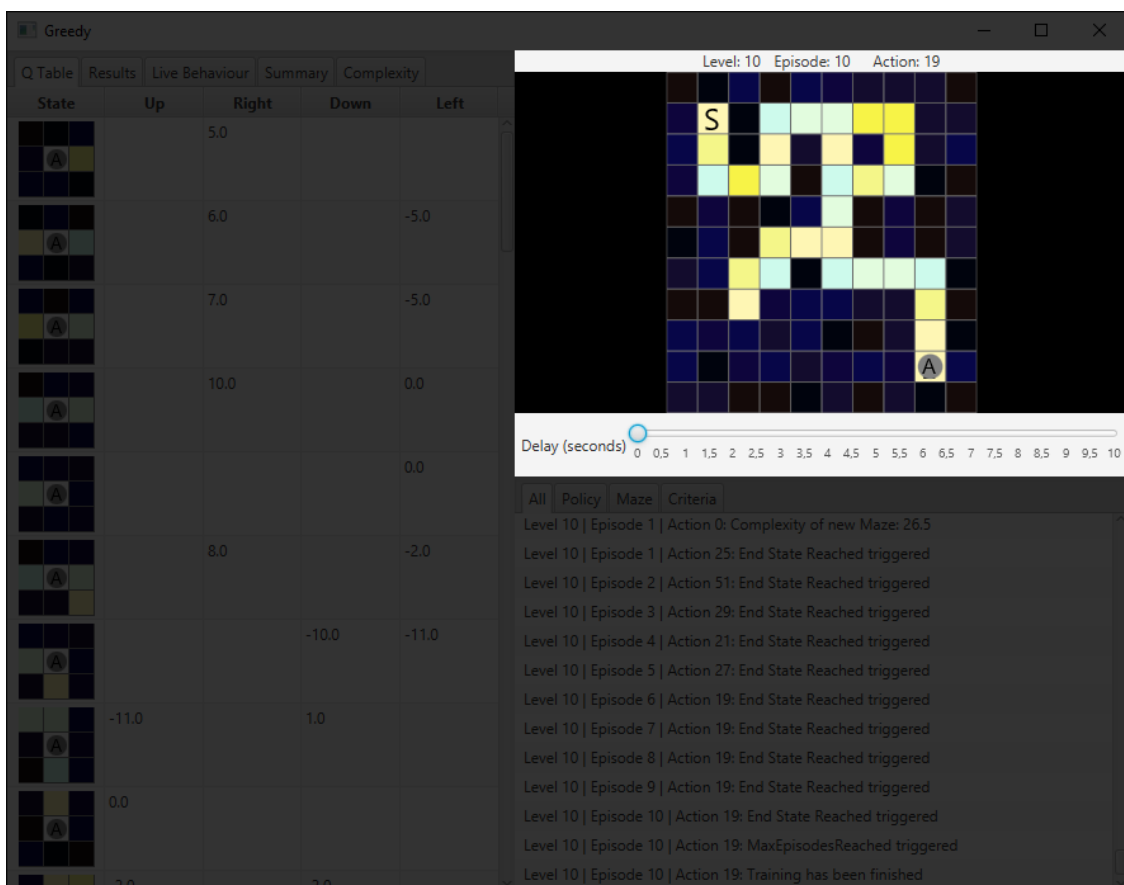


Abbildung 3.1: Simulationsansicht der UI

3.2.2 Infoansicht

In Abbildung 3.2 wird wieder ein Teil der UI hervorgehoben. Dieser hervorgehobene Teil wird im Folgenden als Infoansicht bezeichnet und ist für die anschauliche Aufbereitung der gesammelten Informationen zuständig.

Hierbei werden die aufbereiteten Informationen in sogenannten Tabs platziert. Hintergrund dieser Designentscheidung ist, dass die so dargestellten Informationen leicht erweitert können, indem einfach ein neuer Tab hinzugefügt wird. Derzeit beinhaltet der Info-Tab-Container fünf Elemente.

Der erste Tab ist bereits in der zuvor referenzierten Abbildung 3.2 abgebildet. Hier sieht man eine für Menschen intuitiv verständliche Darstellung der derzeitigen Q-Table des Agenten. Die Zustände, welche intern als lange und recht unverständliche Strings gespeichert werden (siehe Abschnitt 6.1.1), werden hier graphisch dargestellt. Zudem wird in der vorliegenden Tabelle immer der Zustand automatisch selektiert, in welchem sich der Agent derzeit befindet. Somit ist es dem Benutzer jederzeit möglich, in Erfahrung zu bringen, wie die Q-Werte des aktuellen Zustandes aussehen. Die anderen vier Tabs sind in Abbildung 3.3 abgebildet.

In Abbildung 3.3a ist der „Result Tab“ zu sehen. Der Benutzer hat in diesem Tab die Möglichkeit, die Daten einzusehen, die auch in den evaluation.csv Dateien (siehe Abschnitt 8.2.3) gespeichert werden. Der Benutzer sieht hier demnach die tabellarisch aufgelisteten Ergebnisse eines Levels. Hiermit gemeint ist, dass für jede Episode eines Levels angezeigt wird, wie viele Aktionen ein Agent ausgeführt hat und welchen Gesamtreward er letztendlich erhalten hat. Zudem wird für jede Episode angezeigt, welche Bedingung zu seiner Beendigung geführt hat. Der Benutzer kann das Level, zu welchem er diese Informationen angezeigt bekommen möchte, jederzeit ändern, indem er den Spinner über der Tabelle betätigt.

In Abbildung 3.3b wird der „Live Behaviour Tab“ abgebildet. In diesem Tab ist ein Diagramm zu sehen, in welchem sich zwei verschiedene Graphen befinden. Einer dieser Graphen gibt für jede Episode eines Levels die Anzahl der Aktionen an, die der Agent tatsächlich ausgeführt hat. Der andere Graph zeigt hingegen die Anzahl der Aktionen an, die der Agent gebraucht hätte, wenn er den optimalen Pfad genommen hätte. Der Benutzer kann das Level, zu welchem er diese Informationen angezeigt bekommen möchte,

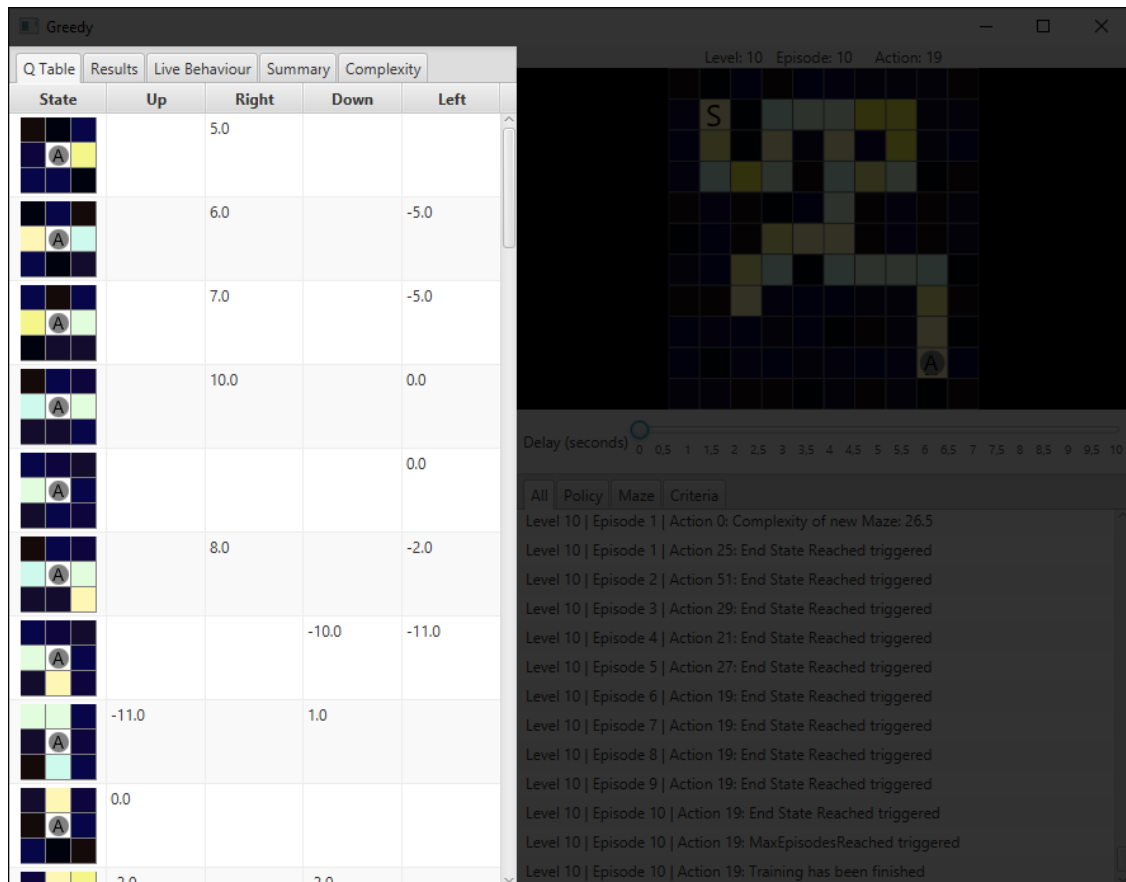


Abbildung 3.2: Infoansicht der UI

auch hier wieder jederzeit ändern, indem er den Spinner über dem Diagramm betätigt.

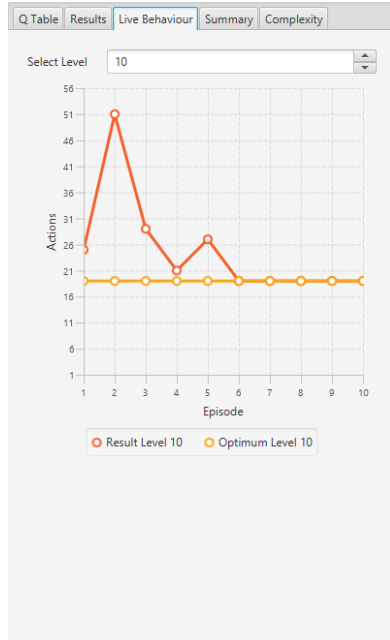
In Abbildung 3.3c wird der „Summary Tab“ dargestellt. In diesem Tab ist wieder ein Diagramm zu sehen, in welchem zwei verschiedene Graphen zu sehen sind. Hier werden allerdings nicht wieder Informationen zu einem bestimmten Level angezeigt, sondern zu allen Levels. Einer der Graphen gibt für jedes Level die Anzahl der Aktionen an, die der Agent gebraucht hätte, wenn er den optimalen Pfad genommen hätte. Der andere Graph gibt hingegen die durchschnittliche Anzahl an Aktionen an, die der Agent im betrachteten Level tatsächlich ausgeführt hat. Mit „durchschnittlicher Anzahl“ ist hierbei der arithmetische Mittelwert der durchgeführten Aktionen aller Episoden eines Levels gemeint.

Der letzte Tab ist der „Complexity Tab“, welcher in Abbildung 3.3d abgebildet ist. Hier wird wieder ein Diagramm gezeigt, welches jedoch nur über einen Graphen verfügt.

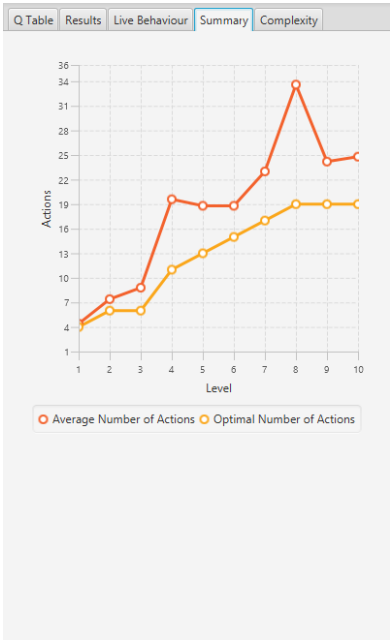
Dieser Graph gibt für jedes Level die Komplexität des verwendeten Labyrinthes an. Die im Graphen abgebildeten Werte entsprechen demnach jenen, die von der verwendeten ComplexityFunction berechnet wurden (siehe Abschnitt 5.3).

[illegible]

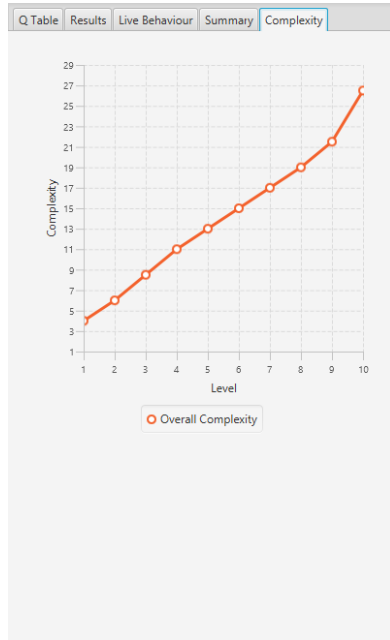
(a) Result Tab



(b) Live Behaviour Tab



(c) Summary Tab



(d) Complexity Tab

Abbildung 3.3: Verschiedene Info Tabs der Simulator UI

3.2.3 UI-Log

Die letzte Hauptkomponente der UI ist das UI-Log, welches in Abbildung 3.4 zu sehen ist. Dieses Log unterscheidet sich grundsätzlich von dem Log, welches in Kapitel 8 vorgestellt wird, da es ausschließlich Textnachrichten speichert, welche zudem nicht persistent gespeichert werden. Die Textnachrichten sind hierbei in verschiedene Typen unterteilt, welche in den entsprechenden Tabs des UI-Logs dargestellt werden. Möchte der Benutzer weitere Typen einführen, muss er lediglich im `GuiMessageType` Enum einen neuen Wert einpflegen. Das UI-Log wird daraufhin beim nächsten Start automatisch um einen weiteren Tab erweitert werden.

Derzeit existieren bereits die folgenden Nachrichtentypen:

- **All:** Hier werden die Nachrichten aller Typen angezeigt.
- **Policy:** Hier werden alle Nachrichten angezeigt, die dem Typ `Policy` zugeordnet wurden. Hier sollten demnach strategiespezifische Informationen geloggt werden. Wird bspw. die Epsilon-Greedy-Strategie (siehe Abschnitt 6.2.3) verwendet, wird hier für jede Aktion angezeigt, ob der Agent eine Greedy- oder eine Random-Aktion getätigt hat.
- **Maze:** Hier werden alle Nachrichten angezeigt, die dem Typ `Maze` zugeordnet wurden. Hier sollten demnach Informationen geloggt werden, die unmittelbar mit dem Labyrinth zusammenhängen. Hier wird bspw. angezeigt, welche Operatoren (siehe Abschnitt 5.2.2) während eines Levelwechsels auf das Labyrinth angewandt wurden.
- **Criteria:** Hier werden alle Nachrichten angezeigt, die dem Typ `Criteria` zugeordnet wurden. Folglich sollten alle Nachrichten, die hier geloggt werden, einen Bezug zu den Bedingungen (siehe Kapitel 7) des Simulators haben. Hier wird bspw. angezeigt, welche Bedingungen zur Beendigung eines Levels oder einer Episode geführt haben.

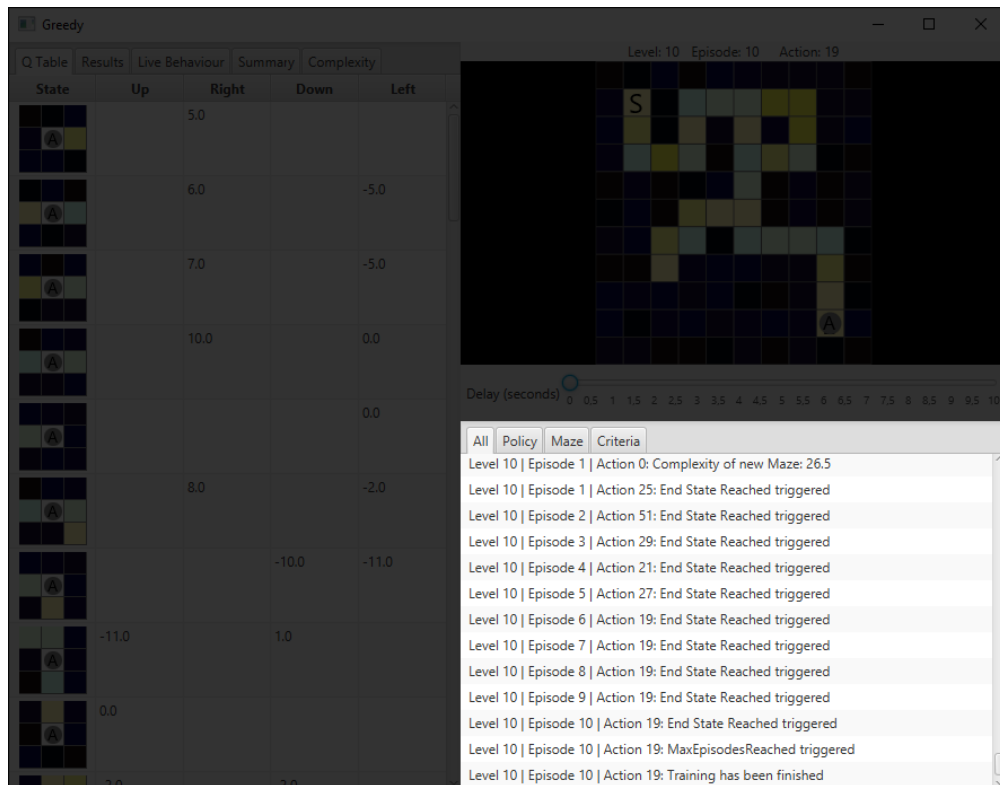


Abbildung 3.4: UI-Log

3.3 Konfigurationsassistent

Über den sogenannten Konfigurationsassistenten ist es möglich, eine Konfigurationsdatei (Siehe Kapitel 4 für mehr Details) durch die Nutzung einer graphischen Oberfläche zu erstellen. Zudem ist es möglich, bereits vorhandene Konfigurationsdateien in die Oberfläche zu laden und das Training direkt über den Assistenten zu starten. In diesem Abschnitt werden zuerst die Benutzeroberfläche und die Funktionalität erklärt. Der Aufbau der Nutzeroberfläche hängt von den Feldern und Annotationen der Config-Klasse ab, dieser Zusammenhang wird im Anschluss beschrieben. Außerdem wird darauf eingegangen, wie der Assistent erweitert werden kann.

3.3.1 Konfigurationsansicht

Abbildung 3.5 zeigt die Oberfläche des Assistenten. Dieser ermöglicht es, den Inhalt einer Konfigurationsdatei darzustellen. Die Schlüssel der Konfigurationsdatei sind in mehreren Kategorien unterteilt. Für jede dieser Kategorien existiert ein eigener Reiter in der Oberfläche. Jeder Reiter ist seinerseits in zwei Bereiche aufgeteilt. Die „Simple Configuration“ enthält die Schlüssel, die immer vom Nutzer eingestellt werden sollten (genaueres dazu in Abschnitt 3.3.2). Jeder der Schlüssel enthält einen voreingestellten Standardwert, welcher abhängig vom Datentyp des Schlüssels ist. Bei diesem Standardwert handelt es sich daher weder um einen empfohlenen, noch zwangsweise um einen zulässigen Eingabewert. Die „Advanced Configuration“ enthält die Schlüssel, die vom Nutzer zwar eingestellt werden können, im Allgemeinen aber nicht verändert werden sollten. Diese Eingabefelder werden daher mit den empfohlenen Einstellungen der Konfigurationsdatei `defaultConfigUIConfig.cfg` vorbelegt, welche sich im `resources`-Ordner des Programms befindet.

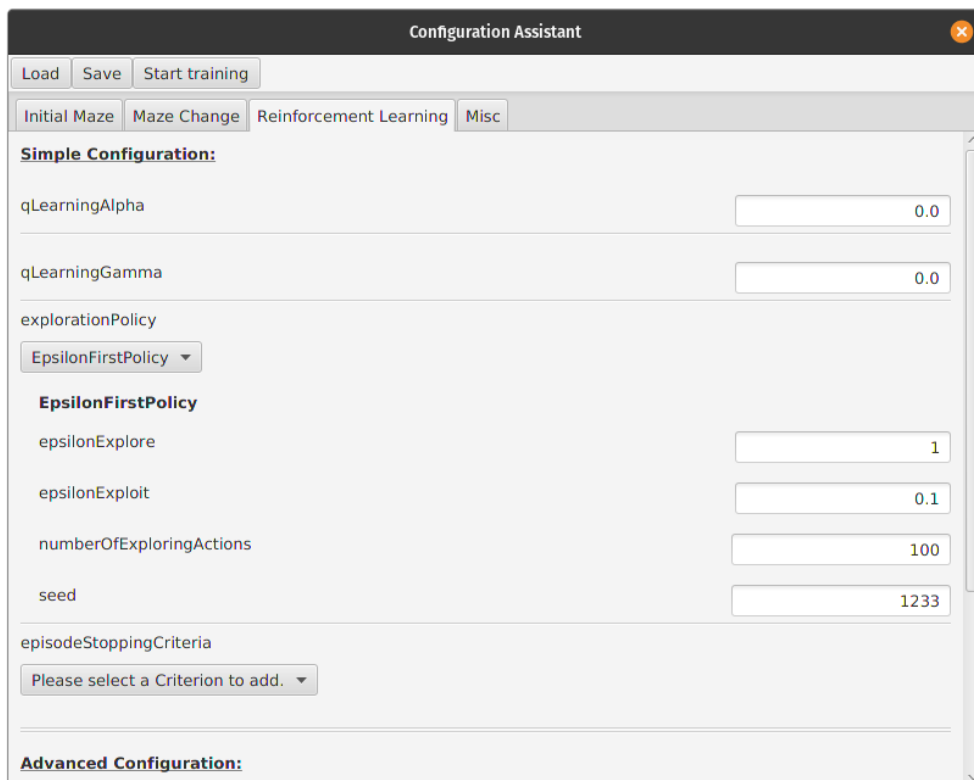


Abbildung 3.5: Nutzeroberfläche des Konfigurationsassistenten

Zusätzlich zu den einfachen Eingabefeldern existieren zudem noch komplexe Eingabefel-

der, welche die Konfiguration komplexer Schlüssel erlauben. Komplexe Schlüssel-Werte sind hierbei, wie in Kapitel 4 erklärt wird, aus mehreren primitiven Schlüssel-Werten aufgebaut. Wie in Abbildung 3.5 z.B. bei der EpsilonFirstPolicy zu sehen ist, sind für diese Werte die Eingabefelder unter einem Namen zusammengefasst. Abbildung 3.6 zeigt, wie diese Werte aus einer vorgegebenen Liste ausgewählt werden können. Im Beispiel sieht man, dass für jede im Programm existierende Explorationsstrategie ein Eintrag vorhanden ist. Wird eine von ihnen ausgewählt, ist es möglich, ihre Parameter in der Nutzeroberfläche einzustellen. Die Eingabefelder werden hierbei wieder mit den von den Datentypen abhängigen Standardwerten vorbelegt. Bei einigen Schlüsseln ist es möglich, mehrere Werte zu konfigurieren. Wie in Abbildung 3.7 zu sehen ist, werden diese ebenfalls unter ihrem jeweiligen Bezeichner zusammengefasst. Einzelne Einträge können mit den hervorgehobenen Knöpfen wieder entfernt werden.

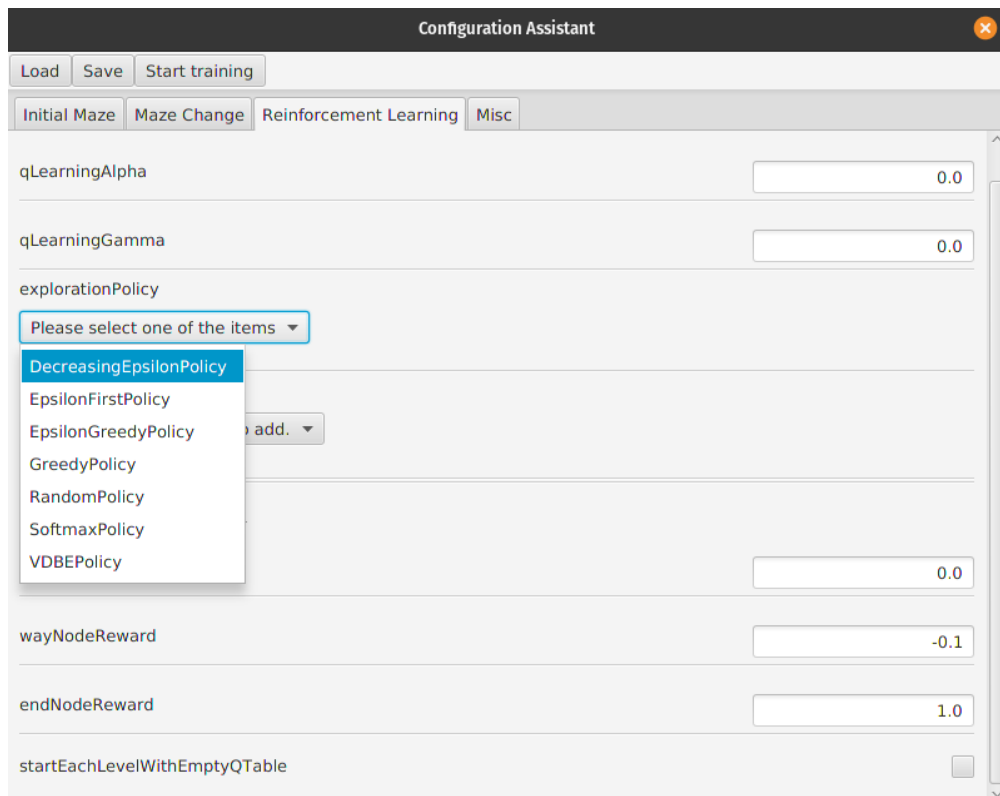


Abbildung 3.6: Auswahl einer Explorationsstrategie im Konfigurationsassistenten

In der oben rechten Ecke der Nutzeroberfläche befinden sich drei Knöpfe. Der Konfigurationsassistent ermöglicht es, bestehende Konfigurationsdateien in die Nutzeroberfläche zu laden, wobei die Datei über einen Dialog ausgewählt werden kann. Außerdem ist es möglich, die derzeitige Konfiguration in einer Datei zu speichern, um sie später für das

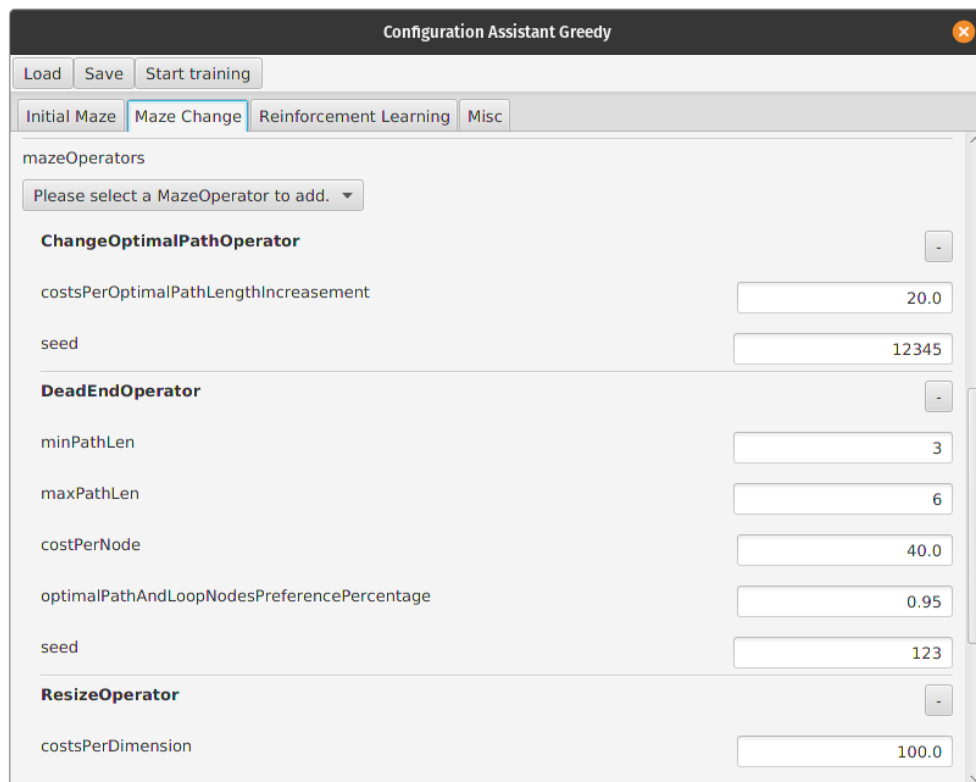


Abbildung 3.7: Liste von Eingabewerten im Konfigurationsassistenten, welche über einen Knopf wieder entfernt werden können

Training nutzen zu können. Zudem ist es ebenfalls möglich, das Training direkt aus dem Konfigurationsassistenten heraus zu starten. Wie in Abbildung 3.8 zu sehen ist, ist es dabei möglich, über einen Dialog zu entscheiden, ob man das Training mit oder ohne die in Abschnitt 3.2 vorgestellte Oberfläche starten möchte. Außerdem kann ein Name für das Training angegeben werden, welcher, wie in Abschnitt 8.1 ausgeführt wird, für den Namen des Logs und der Konfigurationsdatei verantwortlich ist.

Vor jeder der zuletzt aufgeführten Aktionen werden die Eingabewerte geprüft und validiert. Sollte einer der Eingabewerte nicht zulässig sein, wird ein Fehlerdialog angezeigt, der Informationen zu den unzulässig eingestellten Schlüsseln enthält. Außerdem werden diese Fehler auch in der Nutzeroberfläche gekennzeichnet, wie in Abbildung 3.9 zu sehen ist.

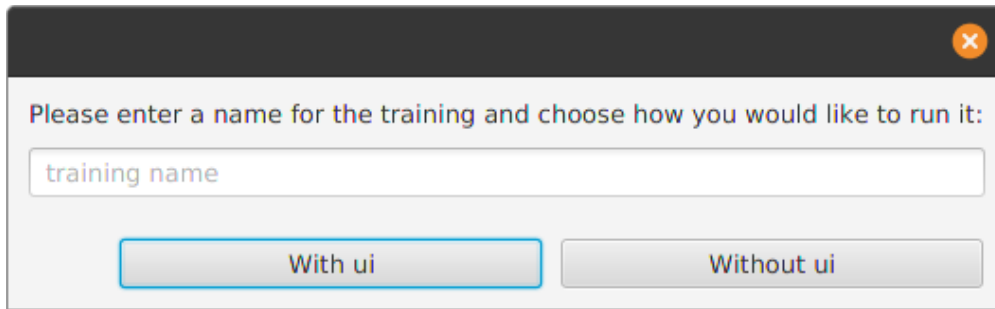


Abbildung 3.8: Dialog zum Start des Trainings im Konfigurationsassistenten

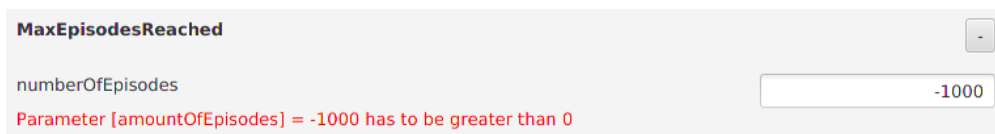


Abbildung 3.9: Fehlermeldung zu einer Eingabe im Konfigurationsassistenten

3.3.2 Erweiterbarkeit und Aufbau

Der Aufbau des Konfigurationsassistenten hängt stark mit der in Kapitel 4 beschriebenen Konfiguration zusammen, für Details sei daher auf diesen Abschnitt verwiesen. Zurzeit unterstützt der Assistent die drei in Abschnitt 4.2.2 genannten Datentypen. Der Assistent stellt die dort beschriebene Config-Klasse graphisch dar. Für jede @Section wird in der Oberfläche ein eigener Reiter angelegt und alle mit ihr annotierten Schlüssel werden in diesem Reiter platziert. Schlüssel ohne @Section werden unter dem Reiter „Various“ zusammengefasst. Alle Schlüssel, die zusätzlich mit @DoNotChange annotiert sind, befinden sich jeweils in jedem Reiter unter der Überschrift Advanced Configuration. Werden der Config-Klasse weitere primitive Schlüssel hinzugefügt, werden diese automatisch in der Oberfläche auftauchen, sofern ihr Datentyp unterstützt wird. Werden neue Klassen für die komplexen Schlüssel hinzugefügt, werden diese ebenfalls in der entsprechenden Auswahl auftauchen. Sollen weitere Datentypen für primitive Schlüssel oder neue komplexe Schlüssel hinzugefügt werden, müssen die beiden Klassen `de.uni.ks.gui.configurator.ConfigurationUI` und `de.uni.ks.gui.configurator.UIFactory` entsprechend angepasst werden. Wie in Abschnitt 3.3 angesprochen, werden die Werte aller mit @DoNotChange annotierten Schlüssel beim Start des Assistenten automatisch gesetzt. Daher muss die bereits zuvor erwähnte im resources-Ordner platzierte Konfigurationsdatei um den neuen Schlüssel ergänzt werden.

4 Konfiguration

Das Programm wird mithilfe einer Datei, welche aus Schlüssel-Wert-Paaren besteht, konfiguriert. Der Pfad dieser Datei wird, wie in Abschnitt 3.1 beschrieben, beim Start des Programms als Argument mit übergeben. Dieses Kapitel beginnt daher mit einer Erklärung, wie sich das Programm mithilfe dieser Datei konfigurieren lässt. Anschließend wird zudem skizziert, wie die Konfiguration erweitert werden kann. Hierbei wird sowohl erklärt, wie sich neue Werte zu bereits existierenden komplexen Schlüsseln (z.B. der verwendeten Explorationsstrategie) ergänzen lassen als auch, wie die Konfiguration um gänzlich neue Schlüssel erweitert werden kann.

4.1 Erstellen einer Konfiguration

Das Programm wird mithilfe einer Datei konfiguriert, im Allgemeinen wird diese die Endung `.cfg` haben. Dies ist aber keine unbedingte Voraussetzung, um die Datei mit dem Programm nutzen zu können. Das Listing 4.1 zeigt einen Ausschnitt aus einer Konfigurationsdatei. Eine Übersicht über alle Schlüssel findet sich in Tabelle 9.1 auf Seite 71 im Anhang. Die Paare folgen allgemein dem Muster `schlüssel = wert`. Kommentare werden mit einem Doppelkreuz (`#`) begonnen und füllen immer die komplette Zeile, auch die einzelnen Schlüssel-Wert-Paare stehen immer in einer Zeile. Mehrzeilige Schlüssel-Wert-Paare werden nicht unterstützt. Die Konfiguration ist in zwei Teile unterteilt. Der erste Teil steht am Anfang der Konfigurationsdatei und enthält Werte, die vom Nutzer bearbeitet werden sollen. Der zweite Teil enthält Werte, die zwar konfiguriert werden können, im Allgemeinen aber nicht verändert werden sollten. Die Datei ist aus drei verschiedenen Arten von Schlüssel-Wert-Paaren aufgebaut:

Primitive Schlüssel-Wert-Paare werden genutzt, um ganzzahlige Werte, Dezimalzahlen und boolesche Werte einzustellen. Gültige Eingabewerte für diese Felder werden in Abschnitt 4.2.2 genannt.

Komplexe Schlüssel-Wert-Paare werden genutzt, um mehrere primitive Schlüssel-Wert-Paare zusammenzufassen. Beispielsweise lässt sich die Explorationsstrategie (explorationPolicy) auf diese Weise konfigurieren. Der Wert für diese Paare folgt dem folgenden Muster: Bezeichner(schlüssel1 = wert1, schlüssel2 = wert2, ...).

Listen von Schlüssel-Wert-Paaren werden genutzt, um ihrerseits mehrere komplexe Schlüssel-Wert-Paare zu kombinieren. Dazu zählen z.B. die Operatoren (siehe Abschnitt 5.2.2), die zur Veränderung des Labyrinths genutzt werden (mazeOperators). Der Wert für diese Paare sieht wie folgt aus: komplexer wert1, komplexer wert2,

Alternativ zur direkten Bearbeitung der Konfigurationsdatei kann auch der Konfigurationsassistent (siehe Abschnitt 3.3) genutzt werden, um eine Konfigurationsdatei zu bearbeiten oder komplett neu zu erstellen. Zum Starten des Assistenten siehe Abschnitt 3.1.

4.2 Erweiterung der Konfiguration

In diesem Abschnitt wird beschrieben, wie man die bestehende Konfiguration erweitern kann. Zuerst wird erklärt, wie man neue Werte zu bereits bestehenden komplexen Schlüsseln ergänzt. Anschließend wird beschrieben, wie man die Konfiguration um gänzlich neue Schlüssel erweitert. Für eine Übersicht über die bestehenden Schlüssel siehe Tabelle 9.1 auf Seite 71 im Anhang.

4.2.1 Einfügen neuer Werte

Das Einfügen neuer Werte für komplexe Schlüssel oder Listen komplexer Werte folgt im Allgemeinen demselben Muster. Damit ein neuer Wert eine gültige Eingabe für einen

Listing 4.1: Ausschnitt aus einer Konfigurationsdatei

```
#Initial Maze
horizontal = true
initialPathLength = 5
numberOfWayColors = 5
numberOfWallColors = 5

#Maze Change
levelChangeCriteria = MaxEpisodesReached(numberOfEpisodes = 2)
numberOfLevels = 2
delta = 200.0

#Reinforcement Learning
trainingName = Greedy
qLearningAlpha = 1.0
qLearningGamma = 1.0
explorationPolicy = GreedyPolicy(seed = 1456)
episodeStoppingCriteria = EndStateReached(), MaxActionsReached(maxActions = 100)

#####
# DO NOT CHANGE THE FOLLOWING PART IF YOU DO NOT KNOW WHAT YOU ARE DOING #
#####

#Initial Maze
generatedWayColorsSeed = 8913
generatedWallColorsSeed = 9122
usedWayColorsSeed = 123
usedWallColorsSeed = 345
minWallWayBrightnessDifference = 200

#Maze Change
complexityFunction = DefaultComplexityFunction()
changeMazeSeed = 123
mazeOperators = ChangeOptimalPathOperator(
    costsPerOptimalPathLengthIncrease = 20.0, seed = 12345),
    DeadEndOperator(minPathLen = 3, maxPathLen = 6, costPerNode = 40.0,
    optimalPathAndParallelRoutesPreferencePercentage = 0.95, seed = 123),
    ResizeOperator(costsPerDimension = 100.0, seed = 123), NewPathOperator(
    minPathLen = 4, maxPathLen = 10, costPerNode = 40.0, seed = 123)

#Reinforcement Learning
initialQValue = 0.0
wayNodeReward = -1
endNodeReward = 10.0
startEachLevelWithEmptyQTable = false

#Features
restrictImageSize = true
showProgressBarInConsole = true
```

Schlüssel wird, muss eine neue Klasse hinzugefügt werden. Diese muss die folgenden Anforderungen erfüllen:

- Die Klasse muss in einem bestimmten Package liegen.
- Die Klasse muss eine bestimmtes Interface implementieren.
- Die Klasse hat entweder keinen oder genau einen Konstruktor.
- Die Parameter des Konstruktors haben einen unterstützten Datentyp (siehe Abschnitt 4.2.2).
- Die Klasse unterstützt es, in eine Konfigurationsdatei geschrieben zu werden. (siehe Abschnitt 4.2.2)

Für die zur Zeit existierenden Schlüssel sind die Packages und Interfaces wie folgt:

explorationPolicy	
Interface	de.uni.ks.agent.explorationPolicies.ExplorationPolicy
Package	de.uni.ks.agent.explorationPolicies
complexityFunction	
Interface	de.uni.ks.maze.complexityFunction.ComplexityFunction
Package	de.uni.ks.maze.complexityFunction
episodeStoppingCriteria	
Interface	de.uni.ks.criterion.Criterion
Package	de.uni.ks.criterion.stopEpisode
levelChangeCriteria	
Interface	de.uni.ks.criterion.Criterion
Package	de.uni.ks.criterion.changeLevel
mazeOperators	
Interface	de.uni.ks.maze.utils.mazeOperators.MazeOperator
Package	de.uni.ks.maze.utils.mazeOperators

4.2.2 Einfügen neuer Schlüssel

Innerhalb des Programms werden die eingelesenen Werte der Konfigurationsdatei in der Klasse `de.uni.ks.configuration.Config` gespeichert. Für das Einlesen der Konfiguration ist die Klasse `de.uni.ks.configuration.ConfigManager` verantwortlich. Schlüssel in der Config-Klasse müssen `public` sein, und können falls gewünscht, annotiert werden:

- `@DoNotChange` markiert Schlüssel, die vom Nutzer nicht bearbeitet werden sollten. Diese stehen im erweiterten Teil der Konfigurationsdatei.
- `@Section(name)` wird genutzt um Schlüssel zu gruppieren. In der Konfigurationsdatei werden Schlüssel mit gleichem Namen zusammen aufgeführt.

Soll die Konfiguration um einen neuen primitiven Schlüssel erweitert werden, reicht es aus, ein neues entsprechendes Feld in der Config-Klasse anzulegen. Unterstützt werden zur Zeit die folgenden Datentypen, da wir annehmen, dass diese Typen alle gängigen Anwendungsfälle abdecken:

- Integer
- Double
- Boolean

Sollte es notwendig sein, einen weiteren primitiven Datentypen einzubinden, so muss das Programm an geeigneter Stelle ergänzt werden, dazu zählen unter anderem: `de.uni.ks.configuration.handlers.PrimitiveKeyHandler`, `de.uni.ks.configuration.handlers.HandlerUtils#parseStringToType` und `de.uni.ks.gui.configurator.ConfigurationUI#buidlPrimitiveField`.

Soll die Konfiguration um einen neuen komplexen Schlüssel erweitert werden, empfiehlt es sich, dafür ein neues Package anzulegen und dort ein Interface zu erstellen, welches die benötigten Schnittstellen (i.e. Methoden) definiert. Dieses Interface muss dabei das Interface `de.uni.ks.configuration.WritableToConfig` (siehe Abschnitt 4.2.2) erweitern. Die Config-Klasse kann dann um den entsprechenden Schlüssel erweitert werden:

```
public Interface name;
```

Soll es möglich sein, mehrere Werte einzustellen, empfiehlt es sich, eine Liste zu nutzen:

```
public List<Interface> name = new ArrayList<>();
```

Um einen neuen Schlüssel einlesen zu können, ist es notwendig, für diesen einen eigenen Handler zu implementieren. Als Vorlage dafür können die beiden Handler `de.uni.ks.configuration.handlers.ExplorationPolicyHandler` und `de.uni.ks.configuration.handlers.EpisodeStoppingCriteriaHandler` dienen.

WritableToConfig

Das Interface `de.uni.ks.configuration.WritableToConfig` muss von jeder Klasse implementiert werden, die direkt in die Konfigurationsdatei geschrieben werden soll. Dazu zählen komplexe Werte und Listen komplexer Werte. Die Methode `String myConfigString` wird dabei dafür genutzt, um aus einer Instanz der Klasse einen Eintrag in der Konfigurationsdatei zu erstellen. Wird beispielsweise im Programm eine `ExplorationPolicy` angelegt:

```
ExplorationPolicy policy = new EpsilonGreedyPolicy(0.5, 123);
```

Wobei die Funktion `myConfigString`, wie in Listing 4.2 gezeigt, implementiert ist, dann ist es möglich, über `policy.myConfigString()` die aktuelle Konfiguration der Instanz zu erhalten. In diesem Beispiel also `EpsilonGreedyPolicy(epsilon = 0.5, seed = 123)`. Diesen Eintrag kann der `ConfigManager` dann aus der Konfigurationsdatei einlesen und daraus wieder die Instanz erstellen. Voraussetzung dafür ist, dass die Syntax aus dem Beispiel eingehalten wird.

Listing 4.2: Beispiel Implementierung für `myConfigString`

```
@Override
public String myConfigString() {
    return getClass().getSimpleName() + "("
        + "epsilon = " + epsilon + ", "
        + "seed = " + seed
        + ")";
}
```

5 Labyrinth

Im nun folgenden Kapitel wird erklärt, wie sich das Labyrinth im Laufe des Trainings entwickelt. Dieses Kapitel beginnt daher mit einer Erklärung, wie das initiale Labyrinth aufgebaut ist und welche Konfigurationsmöglichkeiten der Benutzer bei seiner Generierung hat. Anschließend wird erläutert, wie die levelweise Veränderung des Labyrinthes vonstattengeht. Hierbei wird sowohl der Algorithmus beschrieben, welcher die Veränderung koordiniert, als auch die einzelnen Operatoren, welche die Veränderungen am Labyrinth verursachen. Abschließend wird zudem die im Rahmen des Projektes entwickelte Komplexitätsfunktion vorgestellt, dessen Anspruch es ist, die Komplexität eines Labyrinthes in einem numerischen Wert auszudrücken.

5.1 Initiales Level

In diesem Abschnitt wird beschrieben, wie das initiale Level grundlegend aufgebaut ist und welche Möglichkeiten der Benutzer bei seiner Konfiguration hat.

5.1.1 Aufbau

Das initiale Level besteht immer aus passierbaren Weg- und unpassierbaren Wandfeldern. Die Wegfeldern beinhalten immer das Start- und das Endfeld. Diese beiden Felder sind in jedem Initiallevel durch einen geradlinigen Pfad verbunden, welcher aus weiteren passierbaren Felder besteht. Außerdem gilt, dass die äußersten Felder eines Labyrinthes immer Wandfelder sind. Zudem sei an dieser Stelle angemerkt, dass sowohl die Weg- als auch die Wandfelder über eigene Farbpoolen verfügen, welche sich nicht überschneiden. Des Weiteren gilt, dass die Farben der Wandfelder grundsätzlich dunkler sind

als die der Wegfelder. Dieser Umstand hat keinen technischen Hintergrund, sondern soll lediglich bezwecken, dass Weg- und Wandfelder besser voneinander unterscheidbar sind.

5.1.2 Konfiguration

Während der Konfiguration muss der Benutzer bestimmte Entscheidung hinsichtlich des initialen Levels treffen. Mit dem Schlüssel `horizontal = <Boolean>` bestimmt der Benutzer, ob das Initiallevel horizontal oder vertikal verlaufen soll. Zudem muss er entscheiden, wie lang der zwischen dem Start- und dem Endfeld verlaufende Pfad sein soll. Diese Eigenschaft lässt sich mit dem Schlüssel `initialPathLength = <Integer>` konfigurieren und bestimmt letztendlich, wie groß das initiale Level sein wird. In Abbildung 5.1 sind zwei exemplarische Initiallevel abgebildet, welche den Einfluss der zuvor beschriebenen Schlüssel verdeutlichen sollen.

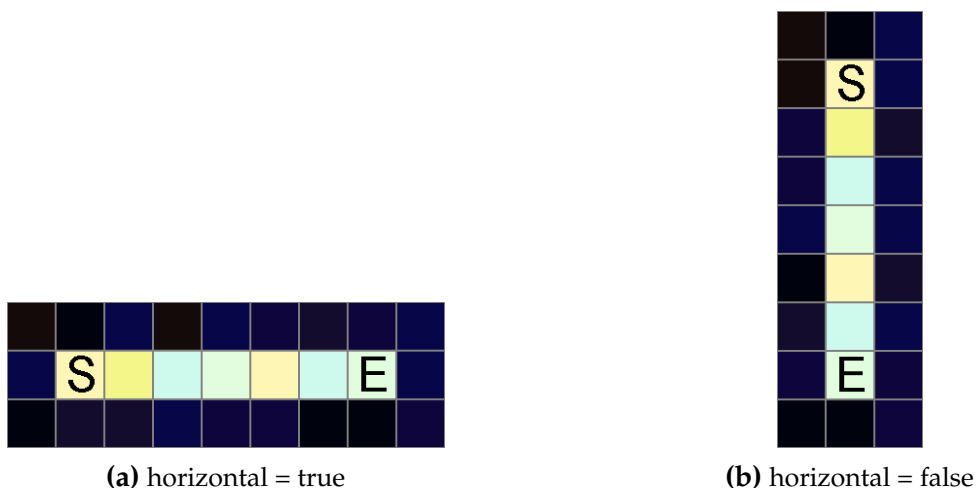


Abbildung 5.1: Horizontales und vertikales Initiallevel mit initialer Pfadlänge 7

Außerdem muss der Benutzer bestimmen, wie viele unterschiedliche Farben jeweils für die Weg- und die Wandfelder verwendet werden sollen. Die Anzahl dieser Farben wird mit den Schlüsseln `numberOfWayColors = <Integer>` und `numberOfWallColors = <Integer>` konfiguriert. Bei der Erstellung der Felder wird diesen immer eine der zu Beginn generierten Farben zugewiesen, wobei Wegfeldern nur Wegfarben und Wandfeldern nur

Wandfarben zugewiesen werden. In Abbildung 5.2 sind zwei Initiallevel mit unterschiedlichen Farbanzahlen exemplarisch abgebildet.

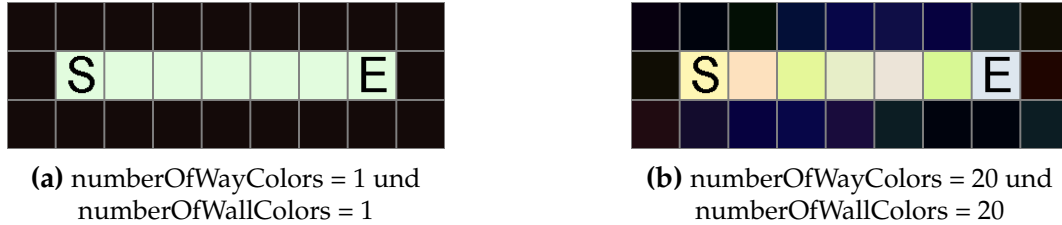


Abbildung 5.2: Initiallevel mit verschiedenen Farbanzahlen

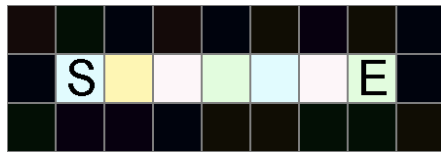
In Abschnitt 5.1.1 wurde erwähnt, dass die Farben der Wandfelder grundsätzlich dunkler sind als die der Wegfelder. Die Helligkeit einer Farbe wird mit der folgenden Formel berechnet:

$$brightness = \sqrt{0.299 * R^2 + 0.587 * G^2 + 0.144 * B^2}$$

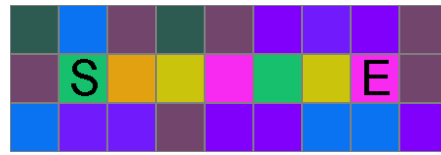
wobei R , G und B jeweils für den entsprechenden Farbanteil im RGB Farbmodell stehen¹. Die Helligkeit kann bei dieser Berechnung einen Wert zwischen 0 und 255 annehmen. Bei der Generierung der Farben werden den Weg- und den Wandfarben zwei unterschiedliche gleich große Helligkeitsintervalle zugewiesen. Der Schlüssel `minWallWayBrightnessDifference = <Integer>` bestimmt hierbei den minimalen Helligkeitsunterschied, den es zwischen der hellsten Wand- und der dunkelste Wegfarbe geben muss. Je höher dieser Wert ist, desto höher wird der Helligkeitsunterschied zwischen den Weg- und den Wandfarben sein. Wichtig ist an dieser Stelle, dass `minWallWayBrightnessDifference` nur Werte zwischen 0 und 255 annehmen kann. In Abbildung 5.3 sind zwei horizontale Initiallevel mit unterschiedlichen `minWallWayBrightnessDifference` Werten zu sehen.

Die Schlüssel `generatedWayColorsSeed = <Integer>` und `generatedWallColorsSeed = <Integer>` werden dazu genutzt, die Zufallsgeneratoren zu initialisieren, die die Farben der Weg- bzw. der Wandfelder generieren. Die Schlüssel `usedWayColorsSeed = <Integer>` und `usedWallColorsSeed = <Integer>` geben jeweils die Seeds der Zufallsgeneratoren an,

¹ Die Berechnungsvorschrift wurde von dieser Seite übernommen: <http://alienryderflex.com/hsp.html>



(a) minWallWayBrightnessDifference = 225



(b) minWallWayBrightnessDifference = 0

Abbildung 5.3: Initiallevels mit unterschiedlichen minWallWayBrightnessDifference Werten

mit welchen die zu Beginn generierten Farben den Weg- und den Wandfeldern zugewiesen werden.

5.2 Levelveränderung

Wie in der Einleitung erwähnt, ist das Training in mehrere Level unterteilt. Wird ein Level durch eine der in Abschnitt 7.2 beschriebenen Bedingungen beendet, so wird das Labyrinth durch den im folgenden Abschnitt erläuterten Algorithmus verändert. Die Anzahl der Level lässt sich in der Konfiguration mit `numberOfLevels = <Integer>` einstellen. Während der Simulation zeigt ein Fortschrittsbalken den prozentualen Anteil der absolvierten Level sowie die verstrichene Zeit in der Konsole an. Dieser lässt sich mit `showProgressBarInConsole = <Boolean>` ein- oder ausschalten.

5.2.1 Algorithmus

Das Labyrinth wird iterativ verändert, bevor der Agent ein neues Level beginnt. Der Grad der Veränderung wird dabei über Δ quantifiziert, welches sich mit dem Schlüssel `delta = <Double>` konfigurieren lässt. Jeder der Operatoren (siehe Abschnitt 5.2.2) verursacht bei seiner Anwendung Kosten. Das Labyrinth wird so lange verändert, bis die Summe der Kosten Δ erreicht oder bis keiner der Operatoren mehr angewandt werden kann. In jeder Iteration wird dabei ein zufälliger Operator ausgewählt und auf das Labyrinth angewandt, sofern die Operation möglich ist und die Kosten noch zulässig sind. Der Zufallsgenerator, der diese Auswahl durchführt, wird mit dem Wert des Schlüssels `changeMazeSeed = <Integer>` initialisiert. Algorithmus 2 skizziert das Vorgehen. Kann

ein Operator in einer Iteration nicht angewandt werden, so wird er aus dem Pool der möglichen Operatoren entfernt. Dadurch terminiert der Algorithmus auch, wenn Δ nicht erreicht werden kann. Da wir annehmen, dass Operatoren, deren Kosten noch erlaubt sind, nur dann nicht genutzt werden können, wenn das Labyrinth für sie zu klein ist, werden nach der Vergrößerung des Labyrinthes wieder alle Operatoren zugelassen (siehe Zeile 8 des Algorithmus). Die hierbei zur Verfügung stehenden Operatoren werden im anschließenden Kapitel erläutert.

Algorithmus 2 Level change algorithm

Require: maze, Δ , operators

```

1: currentDiff = 0
2: while currentDiff <  $\Delta$  & some operator is available do
3:   o = random unmarked operator
4:   if currentDiff + cost of o <  $\Delta$  & o can be used then
5:     currentDiff = currentDiff + cost of o
6:     o changes the maze
7:     if o == ResizeOperator then
8:       unmark all operators
9:   else
10:    mark o as unusable
11: return currentDiff

```

5.2.2 Operatoren

Operatoren werden genutzt, um das Labyrinth zu verändern. Um den Grad der Veränderung zu quantifizieren, besitzt jeder Operator eine Kostenfunktion. Sind die Kosten 0, verändert der Operator das Labyrinth nicht. Je größer die Kosten sind, desto stärker ist auch die Veränderung. Zur Zeit stehen die folgenden Operatoren zur Verfügung:

Vergrößerung: Der *ResizeOperator* vergrößert das Labyrinth und verschiebt das Endfeld. Es gilt im Allgemeinen, dass sich das Startfeld eines Labyrinthes immer in der oberen

linken und das Endfeld in der unteren rechten Ecke des noch zulässigen Teiles des Labyrinthes befinden muss. Diese relative Position muss das Endfeld auch nach der Anwendung des Operators beibehalten, da durch die Anwendung des Operators die optimale Pfadlänge erhöht werden soll. Folglich kann das Labyrinth immer nur nach unten und nach rechts vergrößert werden. Demnach wird das Endfeld bei der Anwendung des Operators immer in die (neue) untere rechte Ecke des Labyrinthes verschoben. Zudem wird, wie in Abbildung 5.4 zu sehen ist, ein entsprechender Weg zwischen dem alten und dem neuen Endfeld angelegt, damit das Endfeld auch nach wie vor erreichbar ist. Die anschließend noch verbleibenden neuen Felder werden als Wandfelder angelegt.

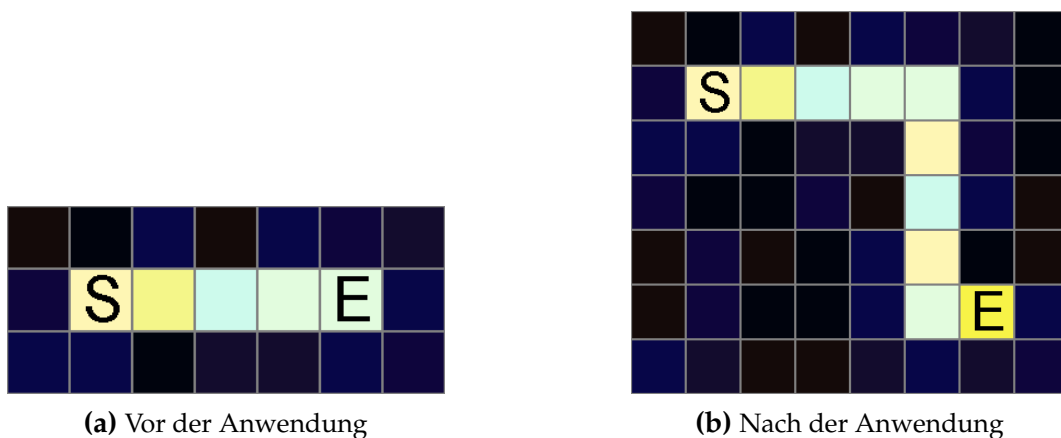


Abbildung 5.4: Anwendung des ResizeOperators

Der Operator lässt sich mit `ResizeOperator(costsPerDimension = <double>, seed = <int>)` einstellen. `costsPerDimension` gibt hierbei an, wie viele Kosten bei einer Erhöhung der X- oder Y-Dimension um 1 entstehen. Im abgebildeten Beispiel waren die erlaubten Kosten 500 und die `costsPerDimension` 100. Die höchstmögliche Erhöhung der Dimensionen war also 5. `seed` initialisiert einen Zufallsgenerator, welcher bestimmt, wie sehr das Labyrinth vergrößert werden soll und wie dieser Vergrößerung auf die zwei Dimensionen aufteilt werden soll. Im abgebildeten Beispiel hat der Zufallsgenerator entschieden, dass die maximal mögliche Vergrößerung (5) vorgenommen werden soll. Zudem entschied er sich, dass die horizontale um 1 und die vertikale Dimension um 4 erweitert werden soll.

Einfügen paralleler Pfade: Der Operator fügt im Labyrinth neue Wege ein, die parallel zum optimalen Pfad verlaufen. Dabei ist der neu entstandene Pfad zwischen zwei Feldern

immer länger als der schon bestehende, sodass kein neuer optimaler Pfad entsteht. Er lässt sich mit `NewPathOperator(minPathLen = <int>, maxPathLen = <int>, costPerNode = <double>, seed = <int>)` einstellen. Der Operator erzeugt dabei Pfade, deren Länge in $[minPathLen, maxPathLen]$ liegt. Ein Zufallsgenerator, der mit `seed` initialisiert wird, beeinflusst sowohl die Lage des Pfades im Labyrinth als auch dessen Verlauf. Die Kosten des Operators sind abhängig von der Länge l des Pfades und berechnen sich nach der Vorschrift $cost = costPerNode * l$. Abbildung 5.5 zeigt eine exemplarische Anwendung des Operators. In diesem Beispiel wurde ein Pfad der Länge 5 eingefügt, sodass die Kosten der Operation $5 * costPerNode$ betragen.

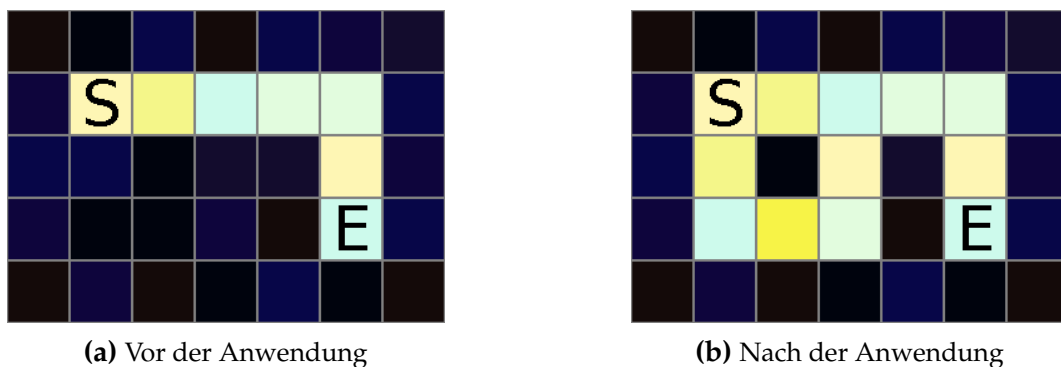


Abbildung 5.5: Anwendung des NewPathOperators

Einfügen von Sackgassen: Der Operator fügt im Labyrinth Sackgassen ein. Diese beginnen an Feldern des optimalen Pfades, eines parallelen Pfades oder an Feldern andere Sackgassen. Mit `DeadEndOperator(minPathLen = <int>, maxPathLen = <int>, costPerNode = <double>, optimalPathAndParallelRoutesPreferencePercentage = <double>, seed = <int>)` wird dieser Operator eingestellt. Die Länge der erzeugten Sackgassen liegt im Intervall $[minPathLen, maxPathLen]$. Ein Zufallsgenerator beeinflusst die Lage der erzeugten Sackgasse im Labyrinth sowie deren Aussehen. Dieser wird mit `seed` initialisiert. Die Kosten des Operators sind wieder abhängig von der Länge l der Sackgasse und berechnen sich nach der Vorschrift $cost = costPerNode * l$. Die Lage der generierten Sackgassen lässt sich mit `optimalPathAndParallelRoutesPreferencePercentage` $\in [0, 1]$ beeinflussen. Je größer der Wert, desto eher werden Sackgassen an den Feldern des optimalen Pfades und an parallelen Pfaden generiert. Für den Wert 0.7 werden also 70% alle Sackgassen an diesen Feldern beginnen. Sollte es nicht möglich sein, weitere Sackgassen

mit dieser Präferenz zu generieren, wird sie ignoriert und Sackgassen werden zufällig an allen Wegfeldern generiert. Sackgassen werden dabei nicht angrenzend an das Endfeld generiert, da diese nicht vom Agenten erreichbar sind. In Abbildung 5.6 ist die Anwendung des Operators gezeigt. In dem Beispiel wurde eine Sackgasse der Länge 4 erzeugt, sodass die Kosten des Operators hier $4 * costPerNode$ betragen.

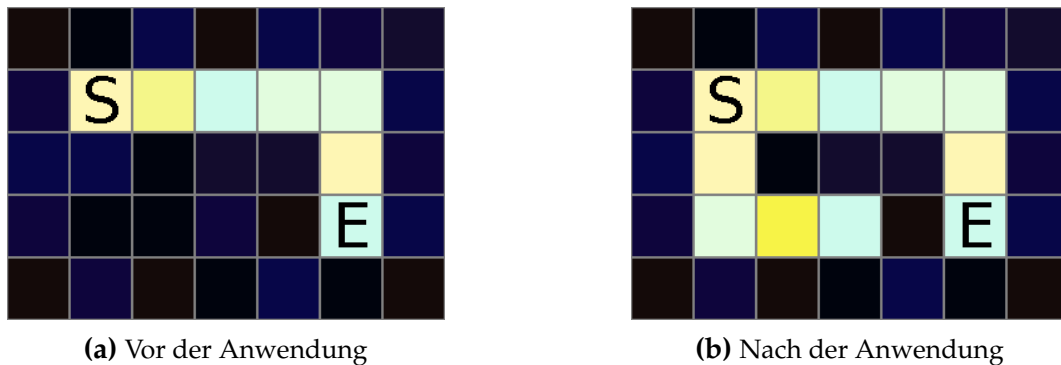


Abbildung 5.6: Anwendung des DeadEndOperators

Verändern des optimalen Pfades: Der `ChangeOptimalPathOperator` verändert den optimalen Pfad des Labyrinthes, indem er eines der auf dem optimalen Pfad liegenden Wegfelder zu einem Wandfeld umwandelt. Ziel dieses „Blockierens“ ist die Verlängerung des optimalen Pfades. In Abbildung 5.7 wird die Anwendung des Operators exemplarisch veranschaulicht. Ein Wegfeld des optimalen Pfades gilt als blockbar, wenn die folgenden Bedingungen erfüllt sind:

- Nach dem Blockieren des Feldes muss ein alternativer Pfad verfügbar sein, über welchen das Endfeld erreichbar ist.
- Nach dem Blockieren ist das Labyrinth immer noch zusammenhängend. Es gilt also, dass jedes passierbare Feld von jedem anderen passierbaren Feld aus erreichbar ist.
- Das zu blockierende Feld ist weder das Start- noch das Endfeld.

Der Operator lässt sich mit `ChangeOptimalPathOperator(costsPerOptimalPathLength-Increasement = <double>, seed = <int>)` einstellen. `costsPerOptimalPathLength-Increasement` gibt hierbei an, wie viele Kosten pro Pfadverlängerung entstehen. Im abgebildeten Beispiel ist die Länge des alten optimalen Pfades 7 und die des neuen 11.

Der neue optimale Pfad hat nun also 4 Wegfelder mehr als der alte. Wäre bspw. der Wert von `costsPerOptimalPathLengthIncrease` 20 gewesen, hätten die Kosten der Anwendung des Operators $20 * 4 = 80$ betragen. `seed` initialisiert einen Zufallsgenerator, welcher bestimmt, welches „blockierbare“ Wegfeld für die Anwendung des Operators gewählt werden soll.

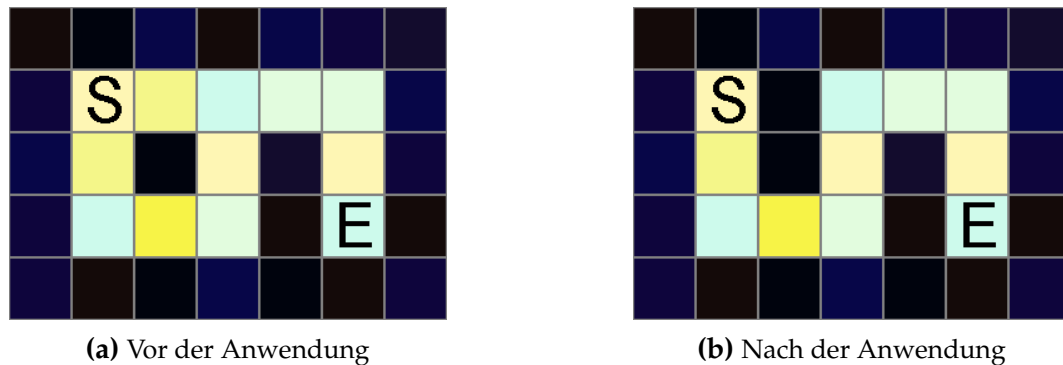


Abbildung 5.7: Anwendung des `ChangeOptimalPathOperators`

5.3 Komplexität

Der Simulator verfügt während jedes Trainings über eine sogenannte Komplexitätsfunktion. Eine solche Funktion bildet ein Labyrinth auf dessen Komplexität ab, welche als Gleitkommazahl angegeben wird. Dies kann hilfreich sein, wenn man beispielsweise die Labyrinthe verschiedener Trainingsdurchläufe miteinander vergleichen möchte. Die Komplexität wird sowohl im Log (siehe Abschnitt 8.2.6) als auch in der Simulator-UI (siehe Abschnitt 3.2.2) angezeigt. Des Weiteren sei an dieser Stelle angemerkt, dass die Komplexitätsfunktion frei konfigurierbar ist. Hierzu muss in der Konfigurationsdatei dem Schlüssel `complexityFunction` der gewünschte Wert zugewiesen werden. Derzeit ist lediglich eine Komplexitätsfunktion implementiert, welche im folgenden Abschnitt beschrieben wird.

5.3.1 DefaultComplexityFunction

Ziel der DefaultComplexityFunction ist es, auszudrücken, wie schwer es für den Agenten sein wird, gute Ergebnisse im übergebenen Labyrinth zu erzielen. Es wurde hierbei versucht, eine Approximation zu finden, die so unabhängig wie möglich von strategiespezifischen Besonderheiten ist. Die Komplexität der DefaultComplexityFunction setzt sich aus der Komplexität dreier Teilaspekte des Labyrinthes zusammen. Im Folgenden wird erklärt, wie die Komplexität dieser einzelnen Teilaspekte berechnet wird und warum sie relevant für die Gesamtkomplexität sind.

Optimale Weglänge Bei der Berechnung der Komplexität wird die Länge des optimalen Weges mit einbezogen. Je länger dieser Weg ist, desto mehr richtige, aufeinanderfolgende Entscheidungen muss der Agent treffen, um den optimalen Reward zu erhalten. Der optimale Pfad und die mit ihm einhergehende optimale Aktionsanzahl wird intern mit dem Dijkstra-Algorithmus berechnet.

In Abbildung 5.8 ist ein Labyrinth zu sehen, dessen optimaler Weg die Länge 5 hat. Der Agent müsste also 4 (aufeinanderfolgende) korrekte Aktionen ausführen, um den höchstmöglichen Reward zu erreichen. Die Anzahl der optimalen Aktionen wird nun mit der im Programm festgelegten Konstante COMPLEXITY_PER_OPTIMAL_ACTION multipliziert, welche derzeit den Wert 1 hat. Daher ist die durch den optimalen Weg erzeugte Komplexität in diesem Beispiel $4 * 1 = 4$.

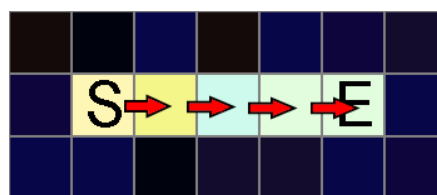


Abbildung 5.8: Beispiel für Berechnung der Komplexität (Weg-Aspekt)

Schleifen Des Weiteren spielen Schleifen bei der Berechnung der Komplexität eine Rolle. Mit Schleifen sind im folgenden Zyklen gemeint, die mindestens zwei Felder besitzen, die auch Teil des optimalen Pfades sind. Grundsätzlich gilt, dass Schleifen das Labyrinth komplexer machen, da der Agent nun mehr Aktionen zur Auswahl hat, die nicht optimal

sind². Des Weiteren ergibt sich bei der zu Beginn definierten Art von Schleifen das Problem, dass der Agent möglicherweise einen Weg vom Start- zum Endfeld erlernt, welcher nicht optimal ist. Dies kann der Fall sein, wenn der Agent einen Parallelweg nimmt, welcher länger als der optimale Weg ist. Wenn die verwendete Explorationsstrategie nun nicht hinreichend explorativ agiert, kann es passieren, dass der optimale Weg nicht mehr gefunden wird.

In Abbildung 5.9 ist ein Labyrinth mit einer Schleife zu sehen. Die Felder, die der Schleife zugeordnet werden können und nicht Teil des optimalen Weges sind, werden gezählt und mit der im Programm festgelegten Konstante `COMPLEXITY_PARALLEL_ROUTE_NODE` multipliziert, welche derzeit den Wert 0.5 hat. Da 5 Schleifen-Felder gezählt werden können beträgt die im abgebildeten Labyrinth durch Schleifen erzeugte Komplexität demnach $5 * 0.5 = 2.5$.

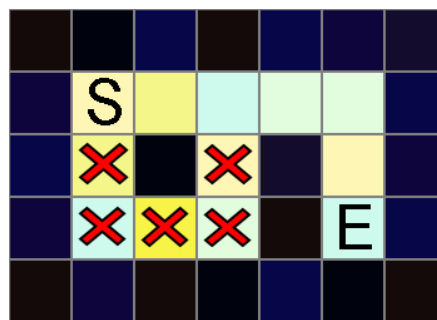


Abbildung 5.9: Beispiel für Berechnung der Komplexität (Schleifen-Aspekt)

Sackgassen haben ebenfalls einen Einfluss auf die Komplexität des Labyrinthes. Dabei spielen sowohl die Länge als auch die Anzahl und Tiefe der Verzweigungen eine Rolle. Allgemein erhöht die Existenz von Sackgassen die Komplexität, da der Agent durch sie mehr Aktionen zur Auswahl hat, die ihn nicht näher zum Endfeld. Zudem gilt, dass das Finden eines Weges umso schwieriger wird, je mehr Aktionen von den einzelnen Feldern aus ausführbar sind. Wenn der Agent während des Trainings eine Sackgasse betritt, muss er diese schnellstmöglich erkennen und verlassen. Ansonsten besteht die Gefahr, dass der Agent das Endfeld nicht mehr erreicht, da er sich möglicherweise in Sackgassen verirren könnte.

In Abbildung 5.10 ist ein Labyrinth abgebildet, auf welches der `DeadEndOperator` mehrmals angewandt wurde. Für die Berechnung der Komplexität werden die Sackgassen

² Das Labyrinth wird zwangsweise komplexer, da es immer nur einen optimalen Weg gibt.

beginnend vom optimalen Pfad aus durchlaufen. Dabei ergibt sich die Komplexität eines Astes aus der Summe der Komplexität seiner Felder. Felder mit zwei Nachbarn haben die im Programm festgelegte Komplexität *COMPLEXITY_DEAD_END_NODE*, die zur Zeit den Wert 1 hat. Die Komplexität der Felder mit drei oder vier Nachbarfeldern wird abhängig von ihrem Verzweigungsgrad v gewichtet:

$$w_v = 1 + (e^{v*0.25} - 1)$$

$$complexity_{node} = w_v * nodeComplexity$$

Die erste Verzweigung in einem Ast hat den Verzweigungsgrad 0, wodurch sich die Formel zu $complexity = 1 * nodeComplexity$ vereinfacht. Felder mit drei Nachbarn haben die Komplexität *COMPLEXITY_THREE_WAY_JUNCTION*, die zur Zeit 3 beträgt. Felder mit vier Nachbarn haben die Komplexität *COMPLEXITY_FOUR_WAY_JUNCTION*, die zur Zeit als 4 festgelegt ist. Die Komplexität des Labyrinthes, bezogen auf die Sackgassen, ergibt sich aus der Summe der Komplexität aller seiner Äste.

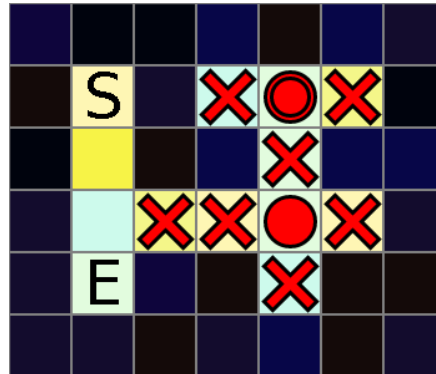


Abbildung 5.10: Beispiel für Berechnung der Komplexität (Sackgassen-Aspekt)

Im Beispiel haben die sieben mit einem x markierten Felder die Komplexität 1. Daraus ergibt sich die Gesamtkomplexität $complexity_x = 7$ für diese Felder. Die erste Verzweigung mit $v = 0$ besitzt vier Nachbarfelder und ist mit einem Kreis gekennzeichnet. Daraus ergibt sich die Komplexität $complexity_4 = 4 * (1 + (e^{0*0.25} - 1)) = 4$ für diese Verzweigung. Die zweite Verzweigung mit $v = 1$ ist mit einem doppelten Kreis gekennzeichnet und besitzt drei benachbarte Felder. Die Komplexität berechnet sich daher wie folgt: $complexity_3 = 3 * (1 + (e^{1*0.25} - 1)) \approx 2.12$. Für die Gesamtkomplexität (in Bezug auf die Sackgassen) ergibt sich damit $complexity_{sum} = complexity_x + complexity_4 + complexity_3 \approx 13.12$.

Sollte der Sonderfall eintreten, dass eine Sackgasse einen Zyklus enthält, z.B. infolge der Anwendung des `ChangeOptimalPathOperator` (siehe Abschnitt 5.2.2), wird diese behandelt, als wären es zwei aufeinander treffende Sackgassen. Da Sackgassen mithilfe von Breitensuche traversiert werden, um die Tiefe einer Verzweigung zu bestimmen, wird die Tiefe in der Reihenfolge bestimmt, in der die Verzweigungen gefunden werden. Die Suche startet dabei bei den unpassierbaren Feldern, die an die Felder des optimalen Pfades angrenzen. Für die Berechnung der Komplexität wird der Zyklus dort in zwei Äste aufgeteilt, wo die Breitensuche auf bereits gesehene Felder trifft.

6 Agent

Im folgenden Kapitel wird erläutert, wie die Zustände im dynamischen Maze-Problem kodiert werden und wie der implementierte Agent lernt. Des Weiteren werden einige Explorationsstrategien, welche im Laufe des Projektes implementiert wurden, skizziert, um dem Benutzer mehr Sicherheit bei ihrer Konfiguration zu geben. Zudem wird erläutert, wie der Simulator um neue Explorationsstrategien erweitern kann.

6.1 Lernverfahren

Im folgenden Abschnitt wird erklärt, wie die Zustände im dynamischen Maze Problem kodiert werden. Zudem wird erläutert, wie das implementierte Lernverfahren (Q-Learning) in den Simulationsablauf eingebunden ist.

6.1.1 Kodierung der Zustände

Wie bereits in Kapitel 2 erwähnt wurde, besitzt der Agent (neben seinem Wissen) zu jedem Zeitpunkt nur Informationen über seine unmittelbare Umgebung. Gemeint ist hiermit, dass der Agent nur weiß, ob die Felder in seiner unmittelbaren Nähe passierbar oder unpassierbar sind und welche Farben sie besitzen. Dieser Umstand wird anhand eines Beispiels in Abbildung 6.1 angedeutet, indem dort der derzeit für den Agent wahrnehmbare Teil des Labyrinthes hervorgehoben wird.

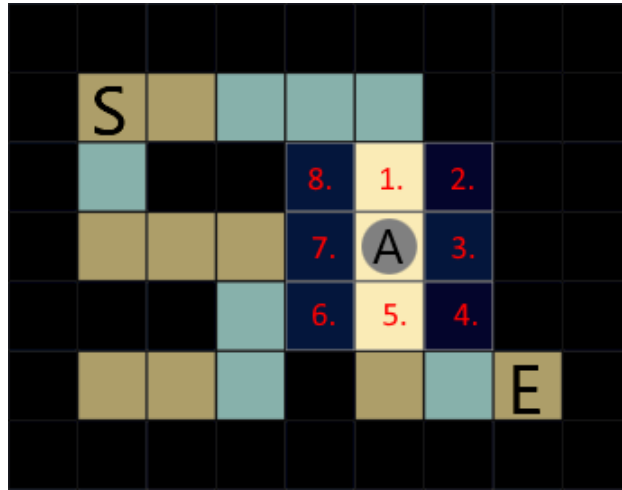


Abbildung 6.1: Zustandskodierung des Agenten

Die zuletzt aufgezählten Informationen der Nachbarfelder müssen in Zuständen zusammengefasst werden, damit ein RL-Verfahren mit dem vorliegenden Problem umgehen kann. Dies wird erreicht, indem die Informationen jedes einzelnen Nachbarfeldes kodiert und in einer Zeichenkette aneinandergereiht werden. Die Reihenfolge, in welcher dies geschieht, lässt sich anhand der in der Abbildung eingezeichneten Nummerierung nachvollziehen. Hierbei kann man ebenfalls erkennen, dass in der Zustandskodierungen keine Informationen zu dem Feld erfasst werden, auf welchem der Agent sich derzeit befindet.

Die Zustandszeichenkette des in Abbildung 6.1 dargestellten Zustandes ist in Listing 6.1 zu sehen. „P“ und „I“ stehen hier jeweils für den Typ des Feldes, wobei „P“ für Passable und „I“ für Impassable steht. Nachdem der Typ angegeben wurde, folgt immer eine Angabe der Farbe des Feldes im RGB Farbraum. Nachdem sowohl der Typ als auch die Farbe eines Feldes angegeben wurde, folgt ein „|“, welches die Beschreibung des nachfolgenden Feldes einleitet.

Listing 6.1: Exemplarische Zustandszeichenkette

```
P[r=250,g=235,b=182] | I[r=4,g=5,b=44] | I[r=4,g=22,b=60] | I[r=4,g=5,b=44] |
P[r=250,g=235,b=182] | I[r=4,g=22,b=60] | I[r=4,g=22,b=60] | I[r=4,g=22,b=60]
```

6.1.2 Q-Learning

In diesem Unterabschnitt wird das Lernverfahren des Agenten, welches auf dem One-Step Q-Learning-Verfahren aufbaut, beschrieben. Ein Schritt dieses Lernverfahrens wird ausgeführt, wenn der Agent eine Aktion tätigen soll (siehe Abschnitt 2.2). Zur Ausführung einer solchen Aktion wurde die sogenannte `doAction`-Methode in der Agent-Klasse implementiert, welche neben der Aktionsauswahl und der Aktionsausführung auch für die Aktualisierung der Q-Werte zuständig ist. Der Ablauf dieser Methode wird in Algorithmus 3 skizziert.

Algorithmus 3 `doAction`-Methode des Agenten

Require: Q-Table q , policy p , node n

- 1: get state s of node n
 - 2: choose action a by executing `chooseAction(s, q)` method of p
 - 3: $Q_{old} \leftarrow Q(s, a)$
 - 4: move agent in direction of action a and observe new node n'
 - 5: get reward r and state s' of new node n'
 - 6: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
 - 7: execute `postProcessing(n, a, Q_{old}, n', q)` method of p
-

Zu Beginn der `doAction`-Methode wird der Zustand s der gegenwärtigen Position n ermittelt (siehe hierzu auch Abschnitt 6.1.1). Anschließend wird die `chooseAction`-Methode der verwendeten Explorationsstrategie p aufgerufen, welche für die Aktionsauswahl zuständig ist und in Abschnitt 6.2.8 genauer beschrieben wird. Nachdem die Aktion a von der Explorationsstrategie p ausgewählt wurde, wird diese vom Agenten umgesetzt, indem der Agent auf die neuen Position n' verschoben wird.

Anschließend wird der Reward r und der neue Zustand s' bestimmt. Der Reward r hängt hierbei allein davon ab, ob es sich bei der neuen Position n' um den Endzustand handelt. Handelt sich bei n' um das Endfeld gilt $r = \text{endNodeReward}$. Gilt jedoch, dass n' lediglich ein reguläres Wegfeld ist, so erhält der Agent den üblicherweise negativen Reward $r = \text{wayNodeReward}$. Anzumerken ist an dieser Stelle, dass `endNodeReward = <Double>` und `wayNodeReward = <Double>` Schlüssel der Konfiguration sind und demnach vom Benutzer in der Konfigurationsdatei eingestellt werden können.

Anschließend wird der Q-Wert des Zustand-Aktions-Paares $Q(s, a)$ nach der Berechnungsvorschrift des One-Step Q-Learnings aktualisiert. Die Lernrate α und der Diskontierungsfaktor γ lassen sich hierbei wieder in der Konfigurationsdatei durch die Schlüssel `qLearningAlpha = <Double>` und `qLearningGamma = <Double>` konfigurieren. Nachdem der Q-Wert aktualisiert wurde, wird die `postProcessing`-Methode der verwendeten Explorationsstrategie p aufgerufen, welche in Abschnitt 6.2.8 genauer beschrieben wird.

Sollte während dieses Ablaufes in der Q-Table q kein Eintrag für den Zustand s oder s' gespeichert sein, wird ein neuer Eintrag erstellt. Hierbei werden Zustand-Aktions-Paare für alle vom entsprechenden Zustand aus ausführbaren Aktionen erstellt, welche mit dem ebenfalls in der Konfigurationsdatei einstellbaren Wert des Schlüssels `initialQValue = <Double>` initialisiert werden.

Es sei zudem an dieser Stelle erwähnt, dass der Benutzer die Möglichkeit hat, den Agenten zu zwingen, nach jedem Level mit einer leeren Q-Table q zu starten, indem er den Schlüssel `startEachLevelWithEmptyQTable = <Boolean>` der Konfigurationsdatei auf `true` setzt.

6.2 Explorationsstrategien

Wie bereits in Abschnitt 6.1.2 erwähnt wurde, bestimmt die verwendete Explorationsstrategie die Aktionsauswahl des Agenten. Einige Explorationsstrategien wurden bereits im Laufe des Projektes implementiert und werden daher an dieser Stelle kurz vorgestellt, um ggf. auftretende Unsicherheiten bei ihrer Parametrisierung zu vermeiden.

6.2.1 Greedy-Strategie

Diese Strategie verhält sich rein exploitativ. Sie wählt immer die Aktion aus, die den höchsten Q-Wert besitzt. Besitzen mehrere Aktionen denselben Q-Wert, so wird zufällig eine von ihnen ausgewählt.

Diese Strategie lässt sich mit `GreedyPolicy(seed = <int>)` einstellen, wobei `seed` den Zufallsgenerator initialisiert, der für die Auswahl der Aktionen zuständig ist, wenn mehrere Aktionen den gleichen Q-Wert besitzen.

6.2.2 Random-Strategie

Diese Strategie verhält sich rein explorativ. Die Aktion wird demnach komplett zufällig gewählt.

Diese Strategie lässt sich mit `RandomPolicy(seed = <int>)` einstellen, wobei `seed` den Zufallsgenerator initialisiert, der für die Auswahl der Aktionen zuständig ist.

6.2.3 Epsilon-Greedy-Strategie

Diese Strategie kann als eine Kombination der Random-Strategie und der Greedy-Strategie angesehen werden. Sie erweitert hierbei die Greedy-Strategie um eine Explorationsrate $\epsilon \in [0, 1]$. Der Wert dieser Explorationsrate bestimmt, mit welcher Wahrscheinlichkeit eine explorative Aktionsauswahl stattfinden soll. Wenn $\epsilon = 1$ gilt, werden die Auswahlwahrscheinlichkeiten analog zur Random-Strategie berechnet. Wenn hingegen $\epsilon = 0$ gilt, verhält sich die ϵ -Greedy-Strategie wie die Greedy-Strategie.

Die Strategie lässt sich mit `EpsilonGreedyPolicy(epsilon = <double>, seed = <int>)` einstellen. Der Parameter `epsilon` repräsentiert hierbei die zuvor beschriebene Explorationsrate ϵ . Für die Initialisierung des Zufallsgenerators wird der Parameter `seed` verwendet. Dieser beeinflusst sowohl die Auswahl des Vorgehens (Greedy oder Random) als auch die Auswahl der Aktionen innerhalb dieser Strategien (siehe Abschnitt 6.2.1 und 6.2.2).

6.2.4 Decreasing-Epsilon-Strategie

Diese Strategie kann als eine Erweiterung der Epsilon-Greedy-Strategie angesehen werden. Der Ansatz dieser Explorationsstrategie ist es, die Wahrscheinlichkeit für eine explorative Aktionsauswahl am Anfang des Trainings deutlich höher als gegen Ende ausfallen zu

lassen. So wird versucht, eine angemessene Exploration des Suchraumes sicherzustellen, bevor der Agent schließlich primär exploitative Aktionsauswahlen tätigt. Dies wird versucht, indem eine anfänglich hohe Explorationsrate $\epsilon_0 \in [0, 1]$ gewählt wird, welche im Laufe des Lernprozesses sukzessiv abnimmt. Diese sukzessive Verkleinerung der anfänglichen Explorationsrate ϵ_0 wird durch einen Abschwächungsfaktor $\lambda \in (0, 1)$ gewährleistet. Für die erste Auswahl der Strategie gilt demnach $\epsilon = \epsilon_0$. Anschließend wird ϵ nach der Formel $\epsilon = \epsilon * \lambda$ neu berechnet. Dieser Berechnung findet nun nach jeder Aktionsauswahl statt.

Die Strategie lässt sich mit `DecreasingEpsilonPolicy(epsilon_0 = <double>, reducingFactor = <double>, seed = <int>)` einstellen. Der Parameter `epsilon_0` repräsentiert hierbei die zuvor beschriebene anfängliche Explorationsrate ϵ_0 . Der Parameter `reducingFactor` beschreibt den Abschwächungsfaktor λ . Der Parameter `seed` wird hier genauso wie bei der Epsilon-Greedy-Strategie (siehe Abschnitt 6.2.3) verwendet.

6.2.5 Epsilon-First-Strategie

Bei dieser Strategie wird das Training in eine Explorationsphase und eine Exploitationsphase aufgeteilt. Die Strategie verhält sich grundlegend wie die Epsilon-Greedy-Strategie, nutzt jedoch für seine zwei Phasen die unterschiedlichen Explorationsparameter $\epsilon_{\text{explore}}$ und $\epsilon_{\text{exploit}}$. Für die ersten n Aktionen verhält sich die Strategie wie die Epsilon-Greedy-Strategie mit dem Explorationsparameter $\epsilon = \epsilon_{\text{explore}}$. Für alle verbleibenden Aktionsauswahlen gilt jedoch für den Explorationsparameter $\epsilon = \epsilon_{\text{exploit}}$. Für die Werte von $\epsilon_{\text{explore}}$ und $\epsilon_{\text{exploit}}$ gelten dieselben Beschränkungen wie für das ϵ der Epsilon-Greedy-Strategie. Zudem sollte $\epsilon_{\text{explore}} \gg \epsilon_{\text{exploit}}$ gelten.

Die Strategie lässt sich mit `EpsilonFirstPolicy(epsilonExplore = <double>, epsilonExploit = <double>, numberOfExploringActions = <int>, seed = <int>)` einstellen. Der Parameter `epsilonExplore` beschreibt hierbei die Explorationsrate der Explorationsphase $\epsilon_{\text{explore}}$. Analog dazu beschreibt der Parameter `epsilonExploit` die Explorationsrate der Exploitationsphase $\epsilon_{\text{exploit}}$. Der Parameter `numberOfExploringActions` beschreibt hierbei die Anzahl der Aktionen n , nach welcher der Phasenwechsel ausgelöst wird. Der Parameter `seed` wird hier wieder wie bei der Epsilon-Greedy-Strategie (siehe Abschnitt 6.2.3) verwendet.

6.2.6 Softmax-Strategie

Ein (mögliches) Problem der Epsilon-Greedy-Strategie ist, dass bei der Berechnung der Auswahlwahrscheinlichkeiten von nicht optimalen Aktionen nicht deren Q-Werte miteinbezogen werden. Alle nicht optimalen Aktionen besitzen daher die gleiche Auswahlwahrscheinlichkeit. Um sich dieses Problems anzunehmen, werden die Auswahlwahrscheinlichkeiten bei der Softmax-Strategie auf der mathematischen Grundlage einer Boltzmann-Verteilung berechnet. Aktionen, die über einen höheren Q-Wert verfügen, besitzen nun eine höhere Auswahlwahrscheinlichkeit. Hierzu wird der Parameter τ , oft als Temperatur-Parameter bezeichnet, eingeführt. Je höher der Wert des Temperatur-Parameter τ ist, desto eher ist die Aktionsauswahl gleichverteilt. Demnach gilt analog zur ϵ -Greedy Strategie, dass die Wahrscheinlichkeit für eine exploitative Aktionsauswahl steigt, je kleiner der Temperatur-Parameter τ ist.

Die Strategie lässt sich mit `SoftmaxPolicy(temperature = <double>, precision = <int>, seed = <int>)` einstellen. Der Parameter `temperature` bezeichnet hierbei den Wert des Temperatur-Parameters τ . Die Berechnungen der Strategie werden aus Präzisionsgründen mit dem `BigDecimal`-Datentyp durchgeführt. Der Parameter `precision` gibt diesbezüglich an, wie viele Ziffern irrationaler Zahlen bei der Anwendung arithmetischer Operatoren herangezogen und gespeichert werden sollen. Mit dem Parameter `seed` wird der Zufallsgenerator der Explorationsstrategie initialisiert.

6.2.7 VDBE-Strategie

Die Value-Difference-Based-Exploration-Strategien (VDBE-Strategien) stellen eine Menge von Explorationsstrategien dar. Wie der Name der Familie vermuten lässt, adaptiert eine VDBE-Strategie die Explorationsparameter einer Explorationsstrategie anhand der Wertunterschiede.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \underbrace{\alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]}_{\text{Wertunterschied}}$$

Im Rahmen des Projektes wurde die VDBE- ϵ -Greedy-Strategie implementiert. Im Gegensatz zur ϵ -Greedy-Strategie wird bei der VDBE- ϵ -Greedy-Strategie keine globale Explorationsrate ϵ verwendet. Stattdessen verfügt bei der VDBE- ϵ -Greedy-Strategie jeder Zustand $s \in S$ über eine eigene Explorationsrate $\epsilon(s) \in [0, 1]$. Der initiale Explorationsparameter eines Zustands wird auch als $\epsilon_0(s)$ bezeichnet. Diese Explorationsrate $\epsilon(s)$ bestimmt analog zur globalen Explorationsrate einer ϵ -Greedy-Strategie die Wahrscheinlichkeit, mit der von einem Zustand s aus eine zufällige Aktion ausgewählt wird. Nach jeder Aktionsauswahl wird die Explorationsrate $\epsilon(s)$ eines Zustandes anhand des Wertunterschiedes adaptiert. Die sogenannte inverse Sensitivität bestimmt hierbei, inwiefern sich der Wertunterschied auf die Adaption des Explorationsparameters auswirkt. Die inverse Sensitivität ist ein Skalierungsparameter und muss demnach in Abhängigkeit von der Höhe der zu erhaltenen Rewards gewählt werden. Sie kann eine beliebige reelle Zahl sein und trägt maßgeblich zum Erfolg oder Misserfolg der Explorationsstrategie bei. Für eine genauere Erklärung sei hiermit auf die Dissertation von von M. Tokic verwiesen¹.

Die Strategie lässt sich mit `VDBEPolicy(inverseSensitivity = <double>, epsilon_0 = <double>, seed = <int>)` einstellen. Der Parameter `inverseSensitivity` gibt hierbei den Wert der zuvor beschriebenen inversen Sensitivität an. Der Parameter `epsilon_0` bestimmt den initialen Epsilon-Wert ϵ_0 eines jeden Zustandes. Der Parameter `seed` hat hier wieder den selben Verwendungszweck wie bei der Epsilon-Greedy-Strategie.

6.2.8 Hinzufügen einer Explorationsstrategie

Das allgemeine Vorgehen für das Hinzufügen neuer Werte ist in Abschnitt 4.2.1 erklärt. Soll eine neue Explorationsstrategie hinzugefügt werden, muss diese in dem dort beschriebenen Package liegen und das Interface `ExplorationPolicy` implementieren. An dieser Stelle soll daher lediglich das Interface genauer beschrieben werden. Dieses besitzt die folgenden zwei wichtigen Funktionen:

chooseAction mit der Signatur `Action chooseAction(Node currentNode, QTable qTable)`. Diese Funktion wird vom Agenten aufgerufen, um eine Aktion auszuwählen (siehe Algorithmus 3). Dafür wird der Funktion sowohl das Feld übergeben, auf dem

¹ Tokic, Michel. Reinforcement Learning mit adaptiver Steuerung von Exploration und Exploitation. Diss. Universität Ulm, 2013.

sich der Agent befindet, als auch dessen aktuelle *Q-Table*. Innerhalb der Funktion kann der Benutzer mit `qTable.getActions(currentNode)` eine Map erhalten, welche als Schlüssel die ausführbaren Aktionen und als Keys die entsprechenden Q-Werte enthält. Der Rückgabewert der Funktion ist die Aktion, die der Agent als nächstes ausführen soll. Die Aktion wird hierbei als Wert eines Enums angegeben, welches sich in `de.uni.k.s.agent.Action` finden lässt.

postProcessing mit der Signatur `default void postProcessing(Node oldNode, Action chosenAction, double oldQValue, Node newNode, QTable qTable)`. Die Funktion bekommt die folgenden Werte übergeben:

- `oldNode`: Das Feld, auf dem der Agent vor der letzten Aktion stand.
- `chosenAction`: Die letzte Aktion, die ausgeführt wurde.
- `oldQValue`: Der alte Q-Wert des Zustand-Aktions-Paares.
- `newNode`: Das Feld, auf dem sich der Agent nach Ausführung der Aktion befindet.
- `qTable`: Die derzeitige Q-Table des Agenten.

Wie im Algorithmus 3 in Abschnitt 6.1.2 skizziert, wird diese Methode aufgerufen, nachdem der Agent eine mit `chooseAction` ausgewählte Aktion durchgeführt hat und der Q-Wert des zugehörigen Zustand-Aktions-Paares aktualisiert wurde. Sie könnte beispielsweise verwendet werden, um auf die Veränderung der Q-Werte zu reagieren. Sie wird z.B. von der VDBE-Strategie verwendet, um die Explorationsrate eines Zustandes anzupassen (siehe Abschnitt 6.2). Die Implementierung dieser Funktion ist optional.

7 Bedingungen

Bedingungen werden innerhalb des Simulators genutzt, um zwei bestimmte Ereignisse auszulösen: den Abbruch einer Episode und die Veränderung eines Levels (siehe auch Abschnitt 2.2). Es ist grundsätzlich möglich, mehrere Bedingungen für diese Ereignisse zu konfigurieren. Es liegt jedoch in der Verantwortung des Anwenders, diese so zu wählen, dass sich keine logischen Widersprüche ergeben. Dieses Kapitel beginnt mit einer Vorstellung der bereits implementierten Episoden- und Levelabbruchbedingungen und schließt mit einer kurzen Anleitung ab, welche skizziert, wie der Simulator um neue Bedingungen erweitert werden kann.

7.1 Episodenabbruch

Episodenabbruchbedingungen werden genutzt, um eine Episode eines Levels abbrechen bzw. zu beenden. In der Konfiguration werden diese über den Schlüssel `episodeStoppingCriteria` konfiguriert. Nach jeder Aktion des Agenten wird überprüft, ob eine Episodenabbruchbedingung erfüllt ist. Tritt eine dieser Bedingungen ein, wird die Episode beendet und der Agent für die nächste Episode vorbereitet. In den folgenden Unterabschnitten werden die bereits implementierten Episodenabbruchbedingungen vorgestellt.

7.1.1 Endzustand erreicht

Die Episode wird beendet, wenn der Agent den Endzustand erreicht hat. Dies ist der Fall, wenn eine Aktion des Agenten dazu geführt hat, dass er sich anschließend auf dem Endfeld des Labyrinthes befindet. In der Konfiguration wird diese Bedingung mit `EndStateReached()` eingestellt.

Diese Bedingung muss nicht zwingend genutzt werden, was dazu führen kann, dass die Simulation anders als erwartet abläuft. Der Hintergrundgedanke hierbei war, dass die Simulation irgendwann um „Zwischenziele“ erweitert werden könnte, welche als Episodenabbruchbedingung implementiert werden könnten. Eine genauere Beschreibung dieser „Zwischenziele“ lässt sich im Ausblick in Abschnitt 9.2 finden.

7.1.2 Maximale Schrittzahl erreicht

Die Episode wird beendet, wenn der Agent eine bestimmte Anzahl an Aktionen im Laufe einer Episode durchgeführt hat. In der Konfiguration wird dies mit `MaxActionsReached(maxActions = <int>)` eingestellt, wobei `maxActions` größer als 0 sein muss.

7.1.3 Optimale Pfadlänge überschritten

Die Episode wird beendet, wenn der Agent eine bestimmte Anzahl an Aktionen getätigt hat, welche in Abhängigkeit von der optimalen Anzahl an Aktionen, die für den Weg vom Start- zum Endfeld notwendig sind, berechnet wird. Hiermit kann erreicht werden, dass eine Episode beendet wird, wenn die Aktionszahl des Agenten die optimalen Aktionsanzahl zu sehr überschreitet. Die Toleranz, die dem Agenten hierbei eingeräumt wird, kann statisch oder prozentual angegeben werden.

Diese Bedingung lässt sich daher auf zwei Arten einstellen:

`AgentExceedsOptimalPathStatic(numberOfExtraActions = <int>)`

oder

`AgentExceedsOptimalPathPercentage(percentageOfExtraActions = <double>)`

Wenn man o als die optimale Aktionsanzahl definiert, berechnet sich die Anzahl x an Aktionen, die der Agent überschreiten muss, damit die statische Variante der Bedingung als erfüllt gilt, wie folgt, wobei `numberOfExtraActions` größer oder gleich 0 sein muss:

$$x = o + \text{numberOfExtraActions}$$

Bei der prozentualen Variante der Bedingung, berechnet sich x wie folgt, wobei `percentageOfExtraActions` ebenfalls größer als 0 sein muss:

$$x = o + (o * \text{percentageOfExtraActions})$$

Beispiel: Möchte man also, dass Episoden abgebrochen werden, wenn der Agent mehr als 10% der eigentlich nötigen Aktionen ausgeführt hat, so kann man dies mit `AgentExceedsOptimalPathPercentage(percentageOfExtraActions = 0.1)` einstellen.

7.2 Levelveränderung

Levelveränderungsbedingungen werden genutzt, um ein Level während des Trainings zu beenden (siehe auch Abschnitt 2.2). In der Konfiguration werden diese über den Schlüssel `levelChangeCriteria` konfiguriert. Nach jeder Episode, die der Agenten absolviert, wird überprüft, ob eine dieser Bedingungen erfüllt ist. Wird ein Level beendet, wird entweder das aktuelle Labyrinth verändert (siehe Abschnitt 5.2) oder das Training wird beendet, falls das letzte Level abgeschlossen wurde. In den folgenden Unterabschnitten werden die bereits implementierten Levelveränderungsbedingungen vorgestellt.

7.2.1 Maximale Episodenanzahl erreicht

Das Level wird beendet, wenn eine bestimmte Anzahl an Episoden absolviert wurde. In der Konfiguration wird diese Bedingung mit `MaxEpisodesReached(numberOfEpisodes = <int>)` eingestellt, wobei der Wert für `numberOfEpisodes` größer als 0 sein muss.

7.2.2 Bestimmte Performance erreicht

Das Level wird beendet, wenn der Agent in einer vom Benutzer festgelegten Anzahl von Episoden höchstens eine bestimmte Maximalanzahl an Aktionen y ausgeführt hat. Für

jede Episode, in der die Aktionsanzahl des Agenten innerhalb des erlaubten Intervalls liegt, wird ein Zähler erhöht. Erreicht dieser Zähler einen bestimmten Wert, wird das Level beendet und der Zähler zurückgesetzt.

Die Maximalanzahl y wird in Abhängigkeit von der optimalen Aktionsanzahl o berechnet. Die dabei dem Agenten eingeräumte Toleranz hinsichtlich der Aktionen, die der Agent zusätzlich zu den zwingend nötigen o Aktionen tätigen darf, kann statisch oder prozentual angegeben werden.

Diese Bedingung lässt sich daher auf zwei Arten einstellen:

```
PerformanceAchievedStaticTolerance(numberOfConsideredEpisodes = <int>,  
numberOfToleranceActions = <int>)
```

oder

```
PerformanceAchievedPercentageTolerance(numberOfConsideredEpisodes = <int>,  
percentageTolerance = <double>).
```

Der Wert `numberOfConsideredEpisodes` steuert dabei die Anzahl der Episoden, in denen der Agent die maximale Anzahl y nicht überschreiten darf und muss daher größer als 0 sein. Die Maximalanzahl y ist immer abhängig davon, wie viele Aktionen mindestens notwendig sind, um das Labyrinth zu durchlaufen, wird jedoch in beiden Varianten unterschiedlich berechnet.

Die Maximalanzahl y berechnet sich bei statischen Variante der Bedingung wie folgt, wobei `numberOfToleranceActions` größer oder gleich 0 sein muss:

$$y = o + \text{numberOfToleranceActions}$$

Bei der dynamischen Variante der Bedingung ist die Berechnungsvorschrift wie folgt, wobei `percentageTolerance` ebenfalls wieder größer oder gleich 0 sein muss.

$$y = o + (o * \text{percentageTolerance})$$

Beispiel: Wenn man annimmt, dass der Agent in einem Level ausreichend gut gelernt hat, wenn er in insgesamt 20 Episoden höchstens 15% mehr Aktionen ausgeführt hat, als im optimalen Fall notwendig wären, so kann man eine Veränderung eines Levels mit der

Bedingung `PerformanceAchievedPercentageTolerance(numberOfConsideredEpisodes = 20, percentageTolerance = 0.15)` herbeiführen.

7.3 Hinzufügen einer Bedingung

Das allgemeine Vorgehen für das Hinzufügen neuer Werte ist in Abschnitt 4.2.1 erklärt. Soll eine neue Bedingung hinzugefügt werden, muss diese im dort genannte Package liegen und das aufgeführte Interface `Criterion` implementieren. An dieser Stelle soll daher lediglich das Interface genauer beschrieben werden. Dieses besitzt die folgenden zwei wichtigen Funktionen:

isMet mit der Signatur `boolean isMet(Training training)`. Diese Methode gibt an, ob eine Bedingung erfüllt ist oder nicht. Die eingestellten Bedingungen werden im Programm dabei iterativ abgefragt. Die Bedingungen für den Abbruch einer Episode werden dabei nach jedem Schritt des Agenten abgefragt, wohingegen die Bedingungen für den Wechsel eines Levels nur nach jeder beendeten Episode abgefragt werden.

reset mit der Signatur `default void reset()`. Diese Funktion wird genutzt, um Bedingungen zurückzusetzen. So wird durch diese Methode beispielsweise bei der Bedingung für die maximale Schrittzahl der Schrittzähler nach dem Ende einer Episode zurückgesetzt. Die Bedingungen für eine Levelveränderung werden nach jedem Level und die Bedingungen für einen Episodenabbruch nach jeder Episode zurückgesetzt. Die Implementation ist optional.

8 Logger

Im nun folgenden Kapitel wird der Logger beschrieben¹. Dieses Kapitel beginnt mit einer Erklärung, wo und unter welchen Namen das Log auf der Festplatte angelegt wird. Anschließend wird konzeptuell und anhand von Beispielen erläutert, welche Daten während des Trainings erfasst und persistent gespeichert werden. Abschließend wird skizziert, wie das Datenmodell des Loggers organisiert ist und wie es gegebenenfalls erweitert werden kann.

8.1 Speicherort

Wenn das Training beendet wird, wird das Log in einem Ordner namens "Logs" gespeichert. Dieser Ordner befindet sich immer in dem Verzeichnis, in welchem auch die Anwendung ausgeführt wurde. Sollte dieser Ordner nicht existieren, wird er für die Speicherung der Logs neu angelegt. In diesem Ordner "Logs" wird anschließend ein neuer Ordner für das Log des zuvor absolvierten Trainings erstellt. Der Name dieses Ordners entspricht dem Wert des Schlüssel `trainingName = <String>` der verwendeten Konfigurationsdatei. Sollte bereits ein Log mit diesem Namen vorhanden sein, wird ein Postfix an den ursprünglichen Bezeichner angehängt.

8.2 Inhalt

Die in Abbildung 8.1 zu sehende Dateistruktur zeigt ein exemplarisches Log, welches nach der Absolvierung eines Trainings mit dem Namen "Greedy" angelegt wurde. Während

¹ Wobei an dieser Stelle anzumerken ist, dass dieses Kapitel sich nur auf das „persistente Log“ bezieht. Neben diesem Log gibt es zudem noch ein nicht persistentes UI-Log, welches in Abschnitt 3.2.3 erklärt wird.

des Trainings wurden zwei Level mit jeweils zwei Episoden absolviert. In den folgenden Unterabschnitten wird konzeptuell und anhand von Beispielen erklärt, welche Daten vom Logger verarbeitet und persistent gespeichert werden. Hierbei wird sich des Öffneren auf die in Abbildung 8.1 zu sehende Dateistruktur bezogen.

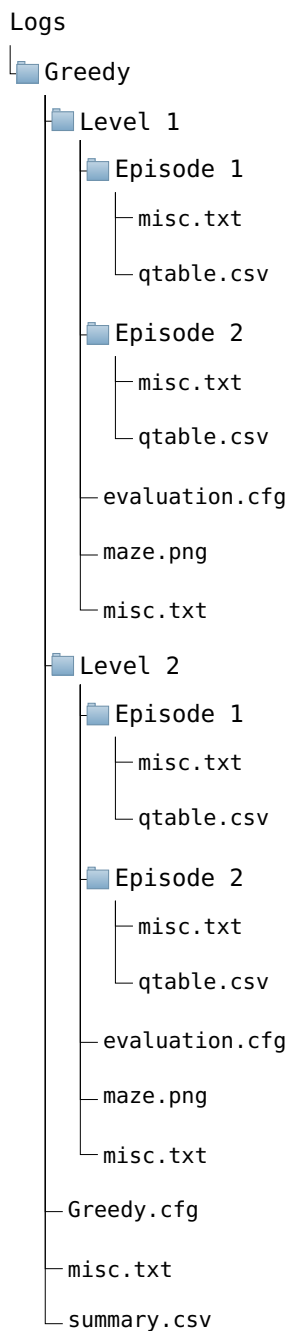


Abbildung 8.1: Dateistruktur eines exemplarischen Logs

8.2.1 miscLog.txt

Es ist während des Trainings jederzeit möglich, gemischte textuelle Informationen zu loggen. Textuelle Information bedeutet hierbei, dass jederzeit ein Textzeile zu einer zukünftigen misc.txt Datei hinzugefügt werden kann. Eine solche Information muss während des Loggens immer einem Objekt aus dem Logger-Datenmodell (siehe Abschnitt 8.3) zugeordnet werden, da diese Zugehörigkeit die zukünftige Position der misc.txt Datei in der Dateistruktur des Logs bestimmt.

Sollte beispielsweise während des Trainings der miscLog von Episode 1 von Level 2 beschrieben werden, wird nach dem Training im Ordner Logs/Greedy/Level 2/Episode 1 eine Datei mit dem Namen misc.txt angelegt, welche beispielsweise wie in Listing 8.1 aussehen könnte. Zudem sind in Listing 8.2 und Listing 8.3 exemplarische misc.txt Dateien eines Levels und eines Trainings dargestellt.

Listing 8.1: misc.txt Datei einer Episode

```
Action 1: Agent moved RIGHT
Action 2: Agent moved RIGHT
Action 3: Agent moved RIGHT
Action 4: Agent moved RIGHT
Action 5: Agent moved DOWN
Action 6: Agent moved DOWN
Action 6: EndStateReached triggered
```

Listing 8.2: misc.txt Datei eines Levels

```
Episode 1: Apply resize operator (x += 2, y += 0)
Episode 2: MaxEpisodesReached triggered
```

Listing 8.3: misc.txt Datei eines Trainings

```
Level 1: Initialised training with config file
Level 1: Start training
Level 2: Apply resize operator (x += 2, y += 0)
Level 2: Finished training
```

8.2.2 qtable.csv

Nach dem Ende jeder Episode wird eine Kopie der derzeit verwendeten QTable gespeichert. Nach Ende des Trainings werden schließlich die Einträge dieser QTables auf der Festplatte gespeichert. Hierfür werden jeweils in den entsprechenden Verzeichnissen der entsprechenden Episoden Dateien mit dem Namen qtable.csv angelegt. In diesen .csv Dateien sind jeweils die folgenden Informationen gespeichert:

- *State*: Gibt die Zustandszeichenkette des Eintrages an (siehe Abschnitt 6.1.1).
- *Up*: Gibt den Q-Wert des Zustand-Aktions-Paars (State, Up) an.
- *Right*: Gibt den Q-Wert des Zustand-Aktions-Paars (State, Right) an.
- *Down*: Gibt den Q-Wert des Zustand-Aktions-Paars (State, Down) an.
- *Left*: Gibt den Q-Wert des Zustand-Aktions-Paars (State, Left) an.

Eine exemplarische QTable ist in Tabelle 8.1 zu sehen. "NaN" bedeutet hierbei, dass für die entsprechenden Aktionen keine Zustand-Aktions-Paare in der QTable existieren. Die Zustandszeichenketten wurden aus Platzgründen ausgelassen, da sie recht lang und für Menschen nur schlecht verständlich sind. Eine exemplarische Zustandszeichenkette lässt sich in Abschnitt 6.1.1 finden.

State	Up	Right	Down	Left
...	NaN	-1.0	NaN	-2.0
...	0.0	NaN	NaN	NaN
...	0.0	NaN	NaN	NaN
...	-1.0	NaN	10.0	NaN
...	NaN	-2.0	NaN	-2.0
...	NaN	-3.0	NaN	NaN
...	NaN	8.0	NaN	-2.0
...	NaN	NaN	-1.0	-2.0
...	NaN	10.0	NaN	0.0
...	NaN	NaN	NaN	0.0

Tabelle 8.1: Inhalt einer qtable.csv Datei

8.2.3 evaluation.csv

Die evaluation.csv Datei fasst die Ergebnisse eines Levels zusammen. Hierzu werden beim persistenten Speichern des Logs für jedes Level die Ergebnisse aller zugehörigen Episoden in einer csv. Datei zusammengefasst. Gespeichert werden in diesen Tabellen die folgenden Daten:

- *Episode*: Gibt die Nummer der Episode an.
- *Number Of Actions*: Gibt die Aktionsanzahl des Agenten am Ende der Episode an.
- *Reward*: Gibt den Reward des Agenten am Ende der Episode an.
- *Episode Stop Criterion*: Gibt die eingetretene Episodenabbruchbedingung an, (siehe Abschnitt 7.1).

Eine exemplarische evaluation.csv Datei ist in Tabelle 8.2 zu sehen.

Episode	Number Of Actions	Reward	Episode Stop Criterion
1	12	-1.0	End State Reached
2	6	5.0	End State Reached

Tabelle 8.2: Inhalt einer evaluation.csv Datei

8.2.4 maze.png

Nach der Beendigung eines Levels wird intern eine Kopie des verwendeten Labyrinthes gesichert. Nach Beendigung des Trainings wird für jedes Level ein Bild des verwendeten Labyrinthes generiert und auf der Festplatte gespeichert. Eine exemplarische maze.png Datei ist in Abbildung 8.2 zu sehen.

Der Benutzer hat bei der Konfiguration die Möglichkeit, die Auflösung der .png Dateien zu beschränken. Hierzu muss der Wert des Schlüssels `restrictImageSize = <Boolean>` auf `true` gesetzt werden. Wird der Wert dieses Schlüssels auf `false` gesetzt, können sehr große Log-Verzeichnisse die Folge sein.



Abbildung 8.2: Eine maze.png Datei

8.2.5 <trainingName>.cfg

Bei der Initialisierung des Loggers wird die für das Training genutzte Konfiguration geloggt. Nach Ende des Trainings wird diese ebenfalls persistent auf der Festplatte gespeichert. Der Bezeichner der hierbei entstehenden Konfigurationsdatei entspricht dem Namen des Schlüssels `trainingName = <String>`, welcher bereits in Abschnitt 8.1 angesprochen wurde. Eine exemplarische Darstellung einer Konfigurationsdatei lässt sich Listing 4.1 finden.

8.2.6 summary.csv

Die Summary.csv fasst die Ergebnisse des kompletten Trainings zusammen. Hierzu werden die hinsichtlich der Leistung des Agenten relevanten Daten jedes Levels in einer Tabelle zusammengefasst. Eine exemplarische summary.csv Datei ist in Tabelle 8.3 beziehungsweise Tabelle 8.4 zu sehen. Gespeichert werden in dieser .csv Datei die folgenden Daten:

- *Level*: Gibt die Nummer des Levels an.
- *Complexity*: Gibt die Komplexität des verwendeten Mazes an (siehe Abschnitt 5.3).
- *Optimal Number Of Actions*: Gibt an, wie viele Aktionen der Agent mindestens gebraucht hätte, um vom Start- zum Endfeld zu gelangen.

- *Average Number Of Actions*: Gibt die Anzahl der Aktionen an, die der Agent durchschnittlich gebraucht hat.
- *Optimal Reward*: Gibt an, welchen Reward der Agent maximal hätte erreichen können.
- *Average Reward* : Gibt an, welchen Reward der Agent durchschnittlich erreicht hat.
- *<Häufigkeit der Episodenabbruchbedingungen>*: Gibt für jede eingestellte Episodenabbruchbedingung (siehe Abschnitt 7.1) an, wie häufig sie eingetreten ist.
- *Level Abort Criterion*: Gibt die Bedingung an, welche zur Beendigung des Levels geführt hat (siehe Abschnitt 7.2).

Level	Complexity	Optimal Number Of Actions	Average Number Of Actions	Optimal Reward	Average Reward
1	4.0	4	6.0	7.0	5.0
2	6.0	6	9.0	5.0	2.0

Tabelle 8.3: 1. Teil einer summary.csv Datei

Level	End State Reached	Max Actions Reached (100 Actions)	Level Abort Criterion
1	2	0	Max Episodes Reached (2 Episodes)
2	2	0	Max Episodes Reached (2 Episodes)

Tabelle 8.4: 2. Teil einer summary.csv Datei

8.3 Funktionsweise

Im folgenden Abschnitt wird die Klassenarchitektur des Loggers beschrieben und es wird skizziert, wie sie erweitert werden kann.

8.3.1 Aufbau

Die geloggten Daten werden in drei Klassen gespeichert:

- TrainingData: Enthält alle Daten, die während eines Trainings geloggt wurden.
- LevelData: Enthält die geloggten Daten eines spezifischen Levels.
- EpisodeData: Enthält die geloggten Daten einer spezifischen Episode.

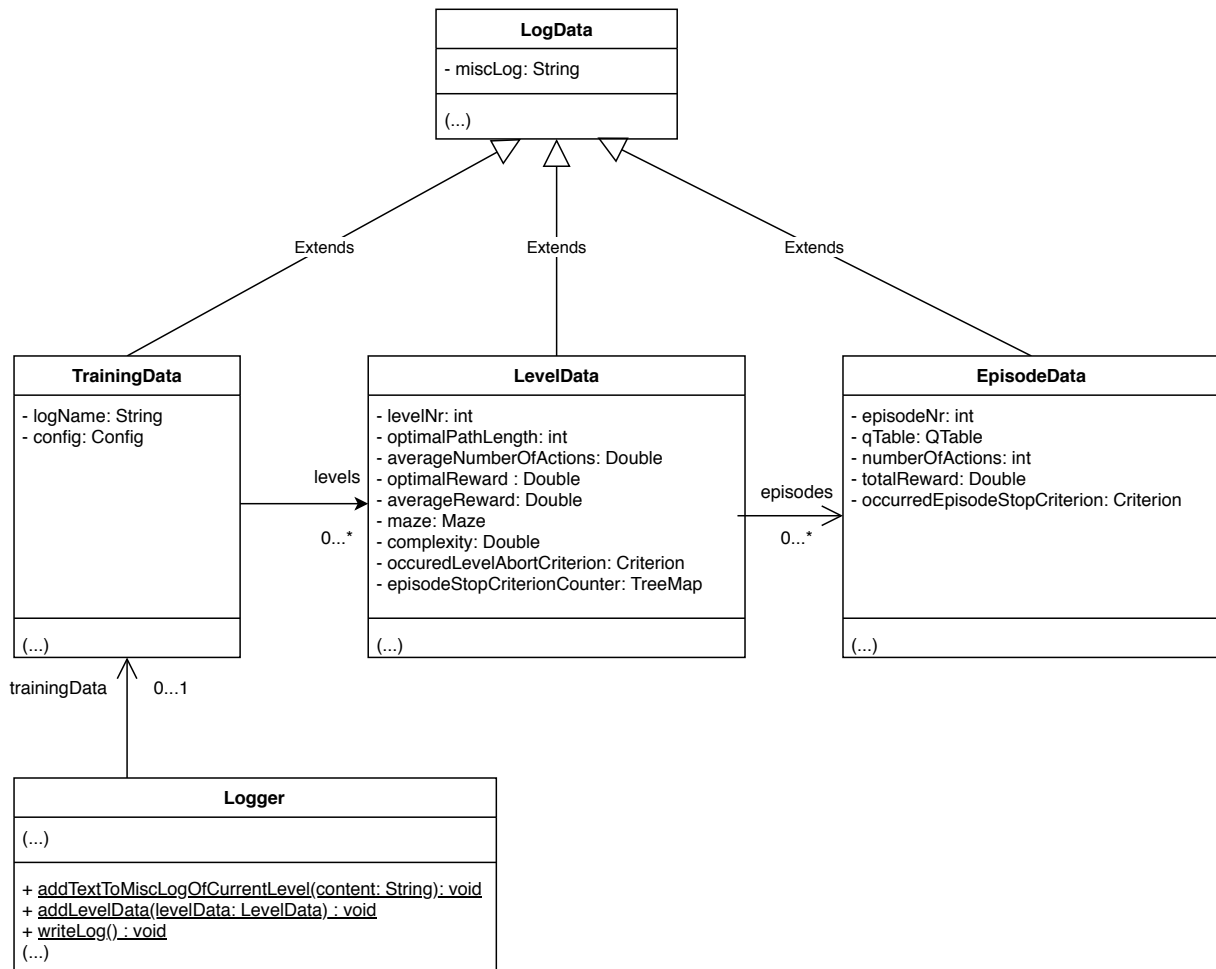


Abbildung 8.3: Inhaltlich und formal vereinfachtes UML-Klassendiagramm des Loggers

Diese drei Klassen erben alle von `LogData`, über welche das sogenannte gemischte Log (siehe Abschnitt 8.2.1) realisiert wird. Die Klasse `Logger` verfügt über (statische) Methoden, welche das Loggen ermöglichen. Nach der Initialisierung besitzt die Klasse `Logger` eine statische Instanz eines `TrainingData` Objektes, in welchem die Daten während des Trainings gespeichert werden. Zudem verfügt der `Logger` über verschiedene Methoden, um die geloggten Daten zu den in Abschnitt 8.2 vorgestellten Dateien zu verarbeiten und persistent zu speichern.

Wie man in Abbildung 8.3 sehen kann, besitzt die Klasse `TrainingData` eine Liste mit `LevelData` Objekten, welche wiederum selbst eine Liste mit `EpisodeData` Objekten besitzen. Zusätzlich zu den `LevelData` Objekten werden in `TrainingData` allgemeine Informationen zum Training gespeichert, welche sich keinem Level bzw. keiner Episode zuordnen lassen. Hierunter fällt z.B. die Konfiguration. Des Weiteren wird in dieser Klasse der Bezeichner vermerkt, unter welchem das Training später auf der Festplatte gespeichert wird.

Objekte der Klasse `LevelData` halten die erfassten Daten eines Levels fest. Neben den `EpisodeData` Objekten enthält ein `LevelData` Objekt zudem weitere für das Level relevante Daten. So wird beispielsweise eine Kopie des verwendeten `Maze` Objektes gespeichert. Zudem können und werden in dieser Klasse bestimmte (eigentlich jederzeit) berechenbare Werte wie beispielsweise die Komplexität des `Maze` Objektes gespeichert. Hierdurch wird versucht, redundante (möglicherweise aufwendige) Berechnungen zu vermeiden.

`EpisodeData` speichert, wie zuvor bereits erwähnt, die Daten einer Episode. Dazu zählt natürlich der erreichte Reward und die Anzahl der durchgeführten Aktionen. Des Weiteren wird hier bspw. die Q-Table, welche der Agent nach Ende der zugehörigen Episode erlernt hat, gespeichert.

8.3.2 Erweiterung

Wenn das Log erweitert werden soll, muss als Erstes festgelegt werden, welcher `Logger` Datenklasse die neuen Daten zugeordnet werden sollen. Daten, die nach jeder Episode geloggt werden müssen, sollten in `EpisodeData` gespeichert werden und Daten, die lediglich nach jedem Level geloggt werden müssen, in `LevelData`. Sollte die Daten sich weder

einer bestimmten Episode noch einem bestimmten Level zuordnen lassen, müssen sie ggf. in `TrainingData` gespeichert werden. Hierbei sollte beachtet werden, dass die Zuordnung im Datenmodell auch den zukünftigen Ort in der Dateistruktur des Logs (siehe Abbildung 8.1 beeinflusst. Gegebenenfalls kann der Benutzer auch abwägen, ob er die neu zu erfassenden Daten im gemischten Log der `Logger-Datenklasse` speichern will (siehe Abschnitt 8.2.1), da hierbei der geringste Aufwand anfallen würde.

Nachdem entschieden wurde, in welcher `Logger-Datenklasse` die Daten gespeichert werden, muss der Benutzer festlegen, zu welchem Zeitpunkt die Daten geloggt werden sollen.

Anschließend muss der Benutzer die Daten im entsprechenden `EpisodeData`-, `LevelData`- oder `TrainingData`-Objekt speichern. Auf dieses kann beispielsweise über die statischen Methoden der Klasse `Logger` zugegriffen werden.

Daraufhin muss der Benutzer noch Code ergänzen, um die neu geloggten Daten persistent auf der Festplatte zu speichern. Hierfür muss eine passende Stelle in der `writeTrainingLog()`-Methode der Klasse `Logger` gefunden werden. An dieser Stelle muss schließlich der Code platziert werden, welcher die persistente Speicherung der Daten veranlassen wird.


9 Schluss

Im nun folgenden Kapitel wird das Projekt abschließend reflektiert. Dabei werden sowohl die Probleme als auch einige mögliche Erweiterungen des Simulators besprochen.


9.1 Probleme

Gegen Ende der Entwicklung wurden einige Probleme erkannt, die nicht mehr ohne größeren Aufwand behoben werden konnten. Diese werden im nun folgenden Abschnitt kurz skizziert.

Gleitkommazahlen Es wurde festgestellt, dass der Datentyp *double* nicht die optimale Wahl hinsichtlich der internen Verwaltung von Q-Werten ist. Aufgrund der für primitiven Datentypen üblichen Limitierung bei der Darstellung und Verarbeitung von Gleitkommazahlen kann es bei ungünstig gewählten Rewards und Initialisierungswerten zu Präzisionsproblemen kommen. Ein Beispiel für dieses Problem ist in Abbildung 9.1 zu sehen. Anstatt des eigentlich gewollten Q-Wertes 0.85 wurde hier im Laufe des Trainings ein Q-Wert berechnet, der sich diesem Wert zwar annähert, jedoch nicht mit ihm identisch ist.

State	Up	Right	Down	Left
		0.85		

(a) Gewünschtes Verhalten

State	Up	Right	Down	Left
		0.8499999999999999		

(b) Fehlerhaftes Verhalten

Abbildung 9.1: Problem bei der Verwaltung von Q-Werten

Fortschrittsbalken Zudem wurde gegen Ende des Projektes ein Fortschrittsbalken implementiert, welcher dem Benutzer in der Konsole anzeigt, wie weit das Training vorangeschritten ist. Die Logik dieses Fortschrittsbalkens wird in einem separaten Thread ausgeführt und regelmäßig aktualisiert. Läuft das Programm ohne Fehler ab (keine Exceptions o.Ä.), funktioniert der Fortschrittsbalken wie gewünscht und der zugehörige Thread wird beendet, sobald das Training ordnungsgemäß beendet wird¹. Wird das Programm jedoch vorzeitig durch einen „unvorhersehbaren“ Fehler abgebrochen, wird der Fortschrittsbalken-Thread nicht beendet. Dies führt dazu, dass die Anwendung nicht terminiert und der Fortschrittsbalken-Thread möglicherweise weiterhin Ausgaben in der Konsole produziert, wie in Abbildung 9.2 zu sehen ist. Es wird daher empfohlen, den Fortschrittsbalken auszuschalten, wenn Änderungen am Code des Simulators vorgenommen werden (siehe Abschnitt 5.2). Des Weiteren kann es zu Problemen kommen, wenn das Konsolenfenster zu schmal wird. Ähnlich zu dem Problem, welches in Abbildung 9.2 zu sehen ist, kann es passieren, dass der Fortschrittsbalken sich über mehrere Zeilen erstreckt und visuell nicht mehr korrekt dargestellt wird, wenn in der ursprünglichen Zeile nicht mehr genügend Platz vorhanden ist.

```
|=====> | 80% | Running for 00:00:01Exception in thread "main" java.lang.RuntimeException: Dummy Exception.  
    at de.uni.ks.Training.doStep(Training.java:113)  
    at de.uni.ks.Training.doTraining(Training.java:82)  
|=====> | 80% | Running for 00:00:01 at de.uni.ks.MazeSimulator.main(MazeSimulator.java:93)  
|=====> | 80% | Running for 00:00:56
```

Abbildung 9.2: Fehlerhaftes Verhalten des Fortschrittsbalken bei Exceptions

Graphische Darstellung Zudem gibt es noch ein kleineres Problem bei der graphischen Darstellung des Trainings in der Simulator-UI. Aufgrund des Aufbaus der Methode, welche für die Ausführung eines Simulationsschrittes zuständig ist, wird das Ergebnis der letzten Aktion einer Episode i.d.R. nicht mehr graphisch dargestellt. Dieses Problem wird in Abbildung 9.3 gezeigt. Erreicht der Agent also beispielsweise durch eine Aktion das Endfeld, wird die neue Agentenposition (der Agent auf dem Endfeld) nicht mehr eingezeichnet. Stattdessen zeigt das nächste Bild den Agenten auf dem Startfeld der neuen Episode.

¹ Das Training gilt als ordnungsgemäß abgeschlossen, wenn das letzte Level beendet oder das Fenster der Simulator-UI geschlossen wird.

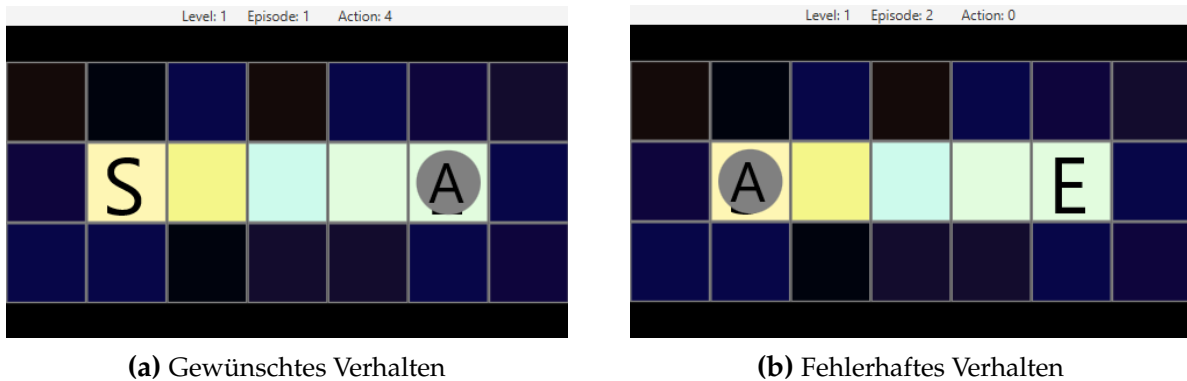


Abbildung 9.3: Verhalten der Simulator-UI bei der letzten Aktion einer Episode

9.2 Ausblick

In diesem Abschnitt wird erläutert, wie der Simulator verbessert werden könnte. Dieser Abschnitt beinhaltet folglich die Vorstellung einiger Lösungsansätze für die im vorausgehenden Abschnitt vorgestellten Probleme. Zudem werden im Laufe des Abschnittes einige mögliche Funktionalitäten beschrieben, um welche der Simulator in Zukunft erweitert werden könnte.

Fehlerbeseitigung Der angesprochene Genauigkeitsverlust bei der Handhabung der Q-Table resultiert daraus, dass diese intern mit `Double` als Datentyp arbeitet. Für eine höhere Genauigkeit (bei arithmetischen Operationen und der Darstellung von Gleitkommazahlen i.A.) könnte die Klasse `BigDecimal`² genutzt werden. Diese ermöglicht es, die Präzision einzustellen, mit welcher die Gleitkommazahlen intern gespeichert und verarbeitet werden. Dabei muss jedoch abgewägt werden, ob die erhöhte Genauigkeit den deutlich höheren Rechenaufwand und die damit einhergehende Verlangsamung des Programms rechtfertigt.

Eine Möglichkeit, das Anzeigen des Fortschrittsbalkens und die Ausgabe von Fehlermeldungen zu vereinen, besteht darin, alle während der Simulation auftretenden Fehlermeldungen und Konsolenausgaben zu sammeln. Diese können dann gegebenenfalls nach Beendigung des Trainings ausgegeben werden. Alternativ kann auch

² Siehe <https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html>

der Thread, der für die Konsolenausgabe zuständig ist, im Fehlerfall beendet werden.

Um den Anzeigefehler der Simulator-UI zu beheben, wäre es notwendig, die Anzeige sofort zu aktualisieren, nachdem der Agent ein Feld betreten hat. Dies müsste passieren, bevor die in Abschnitt 7.1 beschriebenen Bedingungen überprüft werden, bzw. bevor eine Episode beendet wird. Um dies umzusetzen, müsste die Methode, welche einen einzelnen Simulationsschritt durchführt, in geeigneter Weise angepasst werden.

Unterstützung weiterer Datentypen Zurzeit unterstützt das Programm für die Konfiguration lediglich die drei Datentypen Integer, Double und Boolean. Sollen weitere Typen, wie z.B. Byte unterstützt werden, so muss das Programm dafür angepasst werden. Im Abschnitt 4.2.2 wird erläutert, wie eine solche Erweiterung vorgenommen werden könnte.

Farboperator Der Simulator könnte um einen zusätzlichen Labyrinth-Operator erweitert werden, welcher die Farben der Felder verändert. Da die Zustände, mit welchen der Agent arbeitet, sich unter anderem aus den Farben der Nachbarmfelder zusammensetzen (siehe Abschnitt 6.1.1), müsste der Agent sich nach einer solchen Operation in den betroffenen Felder neu orientieren. Die Schwierigkeit beim Implementieren eines solchen Operators besteht darin, sicherzustellen, dass die Operation das Labyrinth komplexer macht. Im Allgemeinen ist es jedoch schwer, eine Aussage über den Zusammenhang der Farbverteilung und der Komplexität eines Labyrinthes zu treffen. Ein Vorteil weniger Farben ist es, dass auch weniger mögliche Zustände existieren und der Agent demnach schneller lernen kann. Allerdings geht hiermit der Nachteil einher, dass der Agent oft die gleichen Zustände sieht und folglich schwer zwischen den eigentlich unterschiedlichen Felder differenzieren kann. Das hat im schlimmsten Fall zur Folge, dass das Lernproblem für den Agenten unlösbar wird. Wenn es viele Farben im Labyrinth gibt, gibt es i.d.R. auch viele mögliche Zustände. Es gilt hierbei, dass der Agent sich umso besser lokalisieren kann, umso mehr erfahrbare Zustände für ihn existieren. Eine hohe Anzahl an Farben sorgt jedoch auch dafür, dass der Agent eine große Q-Table benötigt, um alle Zustände zu verwalten, was wiederum bedeutet, dass Q-Learning länger braucht, um zu konvergieren. Dies hat einen negativen Einfluss auf die Laufzeit des Lernverfahrens. Die Komplexität für den Agenten hängt

neben der verwendeten Explorationsstrategie maßgeblich von der Größe des Labyrinths im Verhältnis zur Anzahl der für ihn erfahrbare Zustände ab. Um einen Farboperator einzufügen, der das Labyrinth komplexer macht, sind also umfangreiche empirische Versuche notwendig. Wie ein neuer Operator hinzugefügt werden könnte, wird in Abschnitt 4.2.2 beschrieben.

Entwicklung besserer Komplexitätsfunktion Das Programm wurde so implementiert, dass es leicht um eine neue Funktion zur Berechnung der Komplexität (siehe Abschnitt 5.3) ergänzt werden kann. Derzeit ist nur eine Komplexitätsfunktion (die `DefaultComplexityFunction`) implementiert, welche jedoch nicht optimal ist, da es an empirischen Daten gefehlt hat, um im Detail aussagen zu können, was ein Labyrinth für einen RL-Lerner kompliziert macht. Es könnte daher beispielsweise versucht werden, eine Komplexitätsfunktion zu entwickeln, welche neben den bereits in der `DefaultComplexityFunction` berücksichtigten Aspekten auch die Anzahl der erfahrbaren Zuständen miteinbezieht.

Veränderung der Level in Abhängigkeit von der Komplexität Wie in Abschnitt 5.2.1 beschrieben, wird der Grad der Veränderung des Labyrinthes derzeit vom Anwender vorgegeben. Es wäre möglich, die Kosten der Operatoren (siehe Abschnitt 5.2.2) stattdessen von ihrem Einfluss auf die Komplexität abhängig zu machen.

Veränderungsgrad dynamisch berechnen Anstatt den Veränderungsgrad vom Nutzer angeben zu lassen, könnte man diesen dynamisch berechnen. Derzeit ist der Wert statisch und gilt demzufolge für alle Veränderungsschritte. Alternativ wäre es möglich, den gewünschten Grad der Veränderung dynamisch durch eine Funktion zu berechnen. Dies könnte beispielsweise in Abhängigkeit von der Größe des Labyrinths oder der Nummer des aktuellen Levels passieren.

Einfügen von Zwischenzielen Zurzeit soll der Agent im Laufe des Trainings einen optimalen Weg vom Start- zum Endfeld finden. Es entsteht ein für RL-Verfahren nur schwer lösbares Problem, wenn diese Aufgabe um den obligatorischen Besuch bestimmter Zwischenzielen erweitert wird. Hiermit ist gemeint, dass der Agent auf dem Weg vom Start- zum Endfeld ein oder mehrere andere Felder (Zwischenziele) passieren muss, um

das Ende einer Episode herbeizuführen. Diese Felder haben dabei gegebenenfalls einen speziellen Reward. Soll die Simulation ein solches Trainingsszenario unterstützen, ist es notwendig, die Methoden zur Bestimmung des optimalen Pfades anzupassen. Derzeit wird der optimale Pfad und der hiermit einhergehende optimale Reward mithilfe des Dijkstra-Algorithmus berechnet. Um Zwischenziele mit in den Pfad einzubeziehen, ist es daher notwendig, den optimalen Pfad auf andere Art und Weise zu bestimmen. Dazu müsste die Klasse `de.uni.k.s.maze.utils.MazeUtils` erweitert werden, in welcher derzeit auch der Dijkstra-Algorithmus implementiert ist.

Übersicht Konfiguration

Schlüssel	Type	Siehe
changeMazeSeed	Integer	Abschnitt 5.2.1
complexityFunction	ComplexityFunction	Abschnitt 5.3
delta	Double	Abschnitt 5.2.1
endNodeReward	Double	Abschnitt 6.1.2
episodeStoppingCriteria	Criterion	Abschnitt 7.1
explorationPolicy	ExplorationPolicy	Abschnitt 6.2
generatedWallColorsSeed	Integer	Abschnitt 5.1.2
generatedWayColorsSeed	Integer	Abschnitt 5.1.2
horizontal	Boolean	Abschnitt 5.1.2
initialQValue	Double	Abschnitt 6.1.2
initialPathLength	Integer	Abschnitt 5.1.2
levelChangeCriteria	Criterion	Abschnitt 7.2
mazeOperators	MazeOperator	Abschnitt 5.2.2
minWallWayBrightnessDifference	Integer	Abschnitt 5.1.2
numberOfLevels	Integer	Abschnitt 5.2
numberOfWallColors	Integer	Abschnitt 5.1.2
numberOfWayColors	Integer	Abschnitt 5.1.2
qLearningAlpha	Double	Abschnitt 6.1.2
qLearningGamma	Double	Abschnitt 6.1.2
restrictImageSize	Boolean	Abschnitt 8.2.4
showProgressBarInConsole	Boolean	Abschnitt 5.2
startEachLevelWithEmptyQTable	Boolean	Abschnitt 6.1.2
trainingName	String	Abschnitt 8.1, 8.2.5
usedWallColorsSeed	Integer	Abschnitt 5.1.2
usedWayColorsSeed	Integer	Abschnitt 5.1.2
wayNodeReward	Double	Abschnitt 6.1.2

Tabelle 9.1: Übersicht über alle Schlüssel der Konfiguration

Übersicht Zuständigkeit

Funktionalität	Verantwortlicher	Siehe
Ablauf der Simulation	Philip Martin	Abschnitt 2.2
Starten der Anwendung	Timo Sturm	Abschnitt 3.1
Simulator-UI	Philip Martin	Abschnitt 3.2
Konfigurationsassistent	Timo Sturm	Abschnitt 3.3
Konfiguration	Timo Sturm	Kapitel 4
Fortschrittsbalken	Timo Sturm	Abschnitt 5.2
Initiales Labyrinth	Philip Martin	Abschnitt 5.1
Algorithmus zur Veränderung des Labyrinthes	Timo Sturm	Abschnitt 5.2.1
Vergrößerungsoperator	Philip Martin	Abschnitt 5.2.2
Parallelwegoperator	Timo Sturm	Abschnitt 5.2.2
Sackgassenoperator	Timo Sturm	Abschnitt 5.2.2
Wegveränderungsoperator	Philip Martin	Abschnitt 5.2.2
Komplexität optimale Weglänge	Philip Martin	Abschnitt 5.3.1
Berechnung des optimalen Pfades	Timo Sturm	Abschnitt 5.3.1
Komplexität Schleifen	Philip Martin	Abschnitt 5.3.1
Schleifenerkennung	Philip Martin	Abschnitt 5.3.1
Komplexität Sackgassen	Timo Sturm	Abschnitt 5.3.1
Kodierung der Zustände	Philip Martin	Abschnitt 6.1.1
Lernverfahren	Philip Martin	Abschnitt 6.1.2
Greedy-Strategie	Philip Martin	Abschnitt 6.1.2
Random-Strategie	Timo Sturm	Abschnitt 6.1.2
Epsilon-Greedy-Strategie	Timo Sturm	Abschnitt 6.1.2
Decreasing-Epsilon-Strategie	Philip Martin	Abschnitt 6.1.2
Epsilon-First-Strategie	Timo Sturm	Abschnitt 6.1.2
Softmax-Strategie	Philip Martin	Abschnitt 6.1.2
VDBE-Strategie	Philip Martin	Abschnitt 6.1.2
Bedingung: Endzustand erreicht	Philip Martin	Abschnitt 7.1.1
Bedingung: Maximale Schrittzahl erreicht	Philip Martin	Abschnitt 7.1.2
Bedingung: Optimale Pfadlänge überschritten	Timo Sturm	Abschnitt 7.1.3
Bedingung: Maximale Episodenanzahl erreicht	Philip Martin	Abschnitt 7.2.1
Bedingung: Bestimmte Performance erreicht	Philip Martin	Abschnitt 7.2.2
Logger	Philip Martin	Kapitel 8

Tabelle 9.2: Auflistung der Funktionalitäten und der verantwortlichen Entwickler