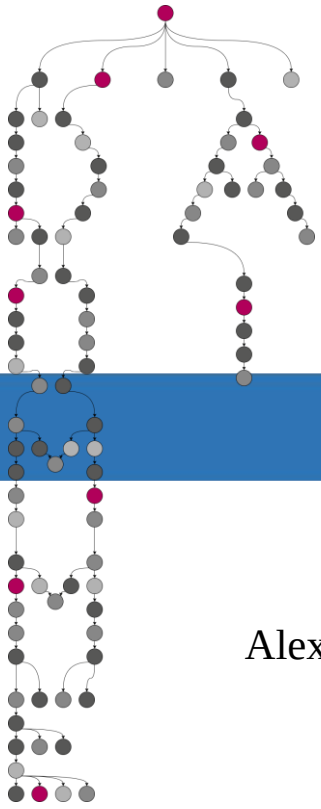
WWU MÜNSTER

informatiCup
Der GI-Wettbewerb für Studierende

DommeAI

Dokumentation & Ausarbeitung
informatiCup 2021



Oktober 2020 - Januar 2021

Alexander Huster – Wigand Kollmeier – Marek Schulte

Inhaltsverzeichnis

Abbildungsverzeichnis iii
Tabellenverzeichnis iii
1 Einleitung 1
2 Dokumentation 2
2.1 Allgemeine Struktur des Codes 2
2.2 Einzelne Methoden und Variablen-Erklärung 5
2.2.1 coord 5
2.2.2 checkCounter/coordIgnore 5
2.2.3 collCounter 5
2.2.4 logActionValue und Gamma 6
2.2.5 calcAction() 6
2.2.6 constructEnemyBoard() 6
2.2.7 checkLeftorRight() 7
2.2.8 checkFront() 7
2.2.9 checkChoices() 8
2.2.10 jobQueue 8
2.3 Strategien 9
2.3.1 checkDeadend() 9
2.3.2 getMinimalEnemyDistance() 11
2.3.3 overSnake() 11
2.3.4 Erläuterung der Strategievariablen 12
2.3.5 Deadend Strategie 13
2.3.6 SafeZone Strategie 14
3 Theoretische Ausarbeitung 14
3.1 Hintergrund 14
3.2 Auswertung & Diskussion 15
4 Softwarearchitektur und -qualität 15
4.1 Software testing 15
4.2 Coding conventions 15
4.3 Wartbarkeit 16
5 Zusätzliche Abgaben 17

5.1 GUI 17
5.2 TensorFlow 18
5.3 Trash-Talk 19
6 Fazit 20
6.1 Ausblick 20
6.2 Schlusswort 20
Literaturverzeichnis 21
Anhang 23

Abbildungsverzeichnis

Abbildung 1: Prozess jobQueue.get 4
Abbildung 2: Ausgangssituation checkDeadend() 9
Abbildung 3: checkDeadend() Teil 1 9
Abbildung 4: checkDeadend() Teil 2+3 10
Abbildung 5: checkDeadend() Teil 4+5 10
Abbildung 6: checkDeadend() Teil 6 10
Abbildung 7: checkDeadend() Teil 7 11
Abbildung 8: Strategiebestimmung 13
Abbildung 9: Beispielansicht eines Spiels 17
Abbildung 10: Klassendiagramm DommeAI 23
Abbildung 11: Schlange in einer Sackgasse vor einem Sprung 24
Abbildung 12: Schlange nach dem Sprung aus einer Sackgasse 24
Abbildung 13: Schlange in einer safeZone 25
Abbildung 14: Ansicht im SciView-Feature 25

Tabellenverzeichnis

Tabelle 1: Erläuterung der für Strategien relevanten Variablen 12

1 Einleitung

Das Team DommeAI besteht aus Alexander Huster, Wigand Kollmeier und Marek Schulte. Wir sind Studenten im fünften Semester des Wirtschaftsinformatik Bachelorstudienganges an der Westfälischen Wilhelms-Universität in Münster.
Während des Studiums haben wir die Programmiersprachen Java und Haskell gelernt und hatten nur sehr kurzen Kontakt mit Python. Aufgrund dessen, dass Python die aktuell beliebteste Programmiersprache ist (Pierre Carbonnelle 2020) und sich auch im beruflichen Kontext immer weiter etabliert (K. R. Srinath 2017), sind wir zu dem Entschluss gekommen, dass es hilfreich wäre, praktische Erfahrungen in der Programmierung mit dieser Sprache zu erlangen und haben die DommeAI daher in Python programmiert.

Unser finaler Ansatz basiert grundlegend darauf, in jedem Zug zu überprüfen, ob die fünf Möglichkeiten - `speed_down`, `change_nothing`, `speed_up`, `turn_left` und `turn_right` - möglich sind. Möglich sind Züge, in denen die Schlange innerhalb der Spielfeldbegrenzungen bleibt, und keine Schlangen, egal ob gegnerische oder eigene, kreuzt. Sonderfälle sind das Beschleunigen und Verlangsamen, da die Spieler eine Geschwindigkeit von 10 nicht überschreiten bzw. 1 nicht unterschreiten dürfen.
Ausgehend von diesen möglichen Zügen werden immer wieder mit den neuen potenziellen States (Koordinaten, Geschwindigkeit, etc.) der Schlangen wiederum rekursiv alle fünf Möglichkeiten geprüft. Unser Ansatz prüft so Ebene für Ebene diese Möglichkeiten, deren Anzahl exponentiell wächst. So sind auf erster Ebene maximal fünf Züge möglich, auf zweiter schon bis zu 25. Nach 7 Ebenen müssen bis zu 78125 Möglichkeiten geprüft werden. Dabei ist zu erwähnen, dass viele Züge natürlich schon eher ausgeschlossen werden, weil z. B. alle Züge, die auf drei Mal `turn_right` hintereinander aufbauen, nicht mehr geprüft werden müssen, da drei Mal `turn_right` in Folge bereits dadurch scheitert, dass die Schlange dann mit sich selbst kollidieren würde.

An dieser Stelle sollte schon erwähnt werden, dass nur Züge ausgeführt werden, welche die maximale berechnete Tiefe potenziell erreichen. Also Züge, die zwar viele Möglichkeiten am Anfang haben, aber dann zum sicheren Ausscheiden führen, werden nicht beachtet.
Außerdem ist ein wichtiger Grundgedanke, dass nur Ebenen betrachtet werden, welche vollständig berechnet wurden. Damit wird verhindert, dass Entscheidungen deswegen bevorzugt werden, weil sie in der Reihenfolge der Berechnungen vorne angeordnet waren.
Während der Programmierung dieses Ansatzes sind uns nach und nach viele verschiedene Optimierungsansätze aufgefallen, die später genauer erläutert werden. Diese Performance-Optimierungen sind notwendig, um eine höhere Tiefe zu erreichen. Die Tiefe ist maßgebend dafür, wie gut unsere Lösung ist, denn je tiefer die Schlange prüfen kann, desto mehr Informationen werden über ihr Umfeld erlangt: Zum Beispiel, ob sie eingeschlossen ist und einen Sprung nutzen muss, um dem Ausscheiden zu entkommen.
Somit ist die Qualität der Entscheidungen der DommeAI sowohl davon abhängig wie effizient der Code ist, als auch wie viel Rechenleistung und Zeit ihr pro Zug zur Verfügung steht. Aufgrund des exponentiellen Wachstums der Anzahl der Möglichkeiten zwischen den Ebenen,

steht der Unterschied in der Entscheidungsqualität auch in einem exponentiellen Zusammenhang mit der Rechenleistung und Zeit pro Zug.

Parallel zu dem erwähnten Ansatz ist noch ein weiterer entstanden, der zwar mit abgegeben wird, jedoch nicht als Lösung, da sich der eben beschriebene Ansatz als besser herausgestellt hat. Es handelt sich dabei um eine Künstliche Intelligenz (KI), die auf einem neuronalen Netz aufbaut und „spe_ed“ erlernt.
Hierfür mussten wir das Spiel nachprogrammieren, um diese KI trainieren zu können, da diese enorm viele Spiele spielen muss, damit ein Fortschritt in der Intelligenz erkennbar ist. Da der Server eine Wartezeit von bis zu fünf Minuten hat, bis ein Spiel beginnt und teilweise bis zu 15 Sekunden Zeit pro Zug erlaubt, ist dieser ungeeignet für das Testen der KI, da es Wochen von Spielen dauern könnte, bis man Fortschritte sehen kann. Ein neuronales Netz arbeitet mit verschiedenen Layern, deren Größe angepasst werden kann. Dies hat Auswirkungen auf die Performance der KI und da es keine exakte Wissenschaft ist, müssen hier viele Parameter getestet werden. Zum Vergleich konnten tausende Spiele pro Sekunde lokal gespielt werden. Dies nur als erste Information, weiter wird noch auf den Ansatz mit einem neuronalen Netz am Ende der Dokumentation unter dem Punkt zusätzliche Abgaben eingegangen.

2 Dokumentation

Im Folgenden wird die Struktur und die Abhängigkeiten der Klassen und Methoden im Allgemeinen erklärt, um das Gesamtkonzept zu verstehen. Anschließend werden alle Methoden und wichtige Variablen genauer erläutert, zum einen deren Funktionsweise, als auch die Notwendigkeit. Außerdem werden die verschiedenen Strategien der DommeAI erklärt und die Methoden, welche für die Strategiebestimmung benutzt werden, werden erläutert.

2.1 Allgemeine Struktur des Codes

Der Start der Software beginnt über die PlaySpe_ed.py, die beim ersten Aufruf die Methode `play()` aufruft und so ein Spiel beginnt. Die Methode bekommt zwei boolesche Ausdrücke entweder übergeben oder nutzt die vordefinierten Initialwerte. `Play()` stellt die Verbindung zum Websocket her und initialisiert den Agenten mit der Höhe und Breite des Feldes, da sich diese nicht während eines Spiels ändern.
Dieser Agent initialisiert eine Job-Warteschlange, sowie weitere Variablen, auf die an dieser Stelle noch kein Fokus gelegt wird. Ausgehend von dem erhaltenen State des Websockets ruft man auf dem Agenten die Methode `gameStep(state)` auf. In `gameStep()` wird jeweils zuerst die Rundenzahl um eins hochgezählt, um später zu sehen, wann ein Sprung stattfinden kann und außerdem wird die Deadline aus dem String in einen timestamp konvertiert, um den Vergleich mit der aktuellen Zeit zu vereinfachen. Dann erstellt `gameStep()` aus dem aktuellen Spielfeld ein neues Brett, in dem alle nächsten Zügen der aktiven Gegner eingetragen sind. Von diesem Brett wird nun ausgegangen, um auf alle Züge des Gegners vorbereitet zu sein, damit Überschneidungen oder ähnliches vermeiden werden.
Danach wird durch `checkDeadend()` geprüft, ob die Schlange sich in einer Sackgasse befindet, worauf im Kapitel der Strategien weiter eingegangen wird. Im Anschluss werden durch

`getDistance()` die Anzahl freier Felder ausgehend von der aktuellen Richtung des Kopfes nach links, rechts und vorne gemessen.
Nun wird die Rekursion begonnen, indem die Methoden `checkFront()` und zwei Mal `checkLeftorRight()` mit den jeweiligen Distanzen, zusätzlich zu den Koordinaten, der Geschwindigkeit und der Richtung aufgerufen werden. Die `action` wird bei `checkFront()` mit `None` übergeben, um die Initialisierung der Züge nach vorne zu ermöglichen. Für links wird eine 3, für rechts eine 4 übergeben, die stellvertretend für diese Aktion stehen.
`CheckFront()` ist eine gebündelte Methode für `slow_down`, `change_nothing` und `speed_up`, da diese teilweise die gleichen Koordinaten auf einen Treffer mit einer Schlange überprüfen müssen und ob das Spielfeld verlassen wird. Somit wird die Überprüfung effizienter, weil keine doppelten Prüfungen geschehen müssen.
Genaueres, wie die Methoden funktionieren wird im nachstehenden Kapitel erläutert. Wichtig an dieser Stelle ist, dass alle drei Methoden, also `checkFront()` und die beiden Male `checkLeftorRight()` die Methode `checkChoices()` in die Warteschlange einfügen.
Nachdem dieser erste Schritt absolviert wurde, beginnt die Abarbeitung der Warteschlange und dies, solange diese nicht leer ist. Wenn `checkChoices()` aus der Warteschlange genommen und ausgeführt wird, werden die fünf möglichen Züge durch `checkFront()` und zwei Mal `checkLeftorRight()` geprüft, ob sie gültig sind und fügen dann wieder `checkChoices()` in die Warteschlange ein. In dieser Methode befindet sich auch ein Abbruchkriterium, dass die Warteschlange leert, wenn weniger als eine vorher festgelegte Zeit bis zur Deadline verbleibt.

Es folgt ein Aktivitätsdiagramm des Ablaufs, der Bearbeitung eines Elements der Warteschlange. Der Prozess ist aus Verständlichkeitsgründen etwas simplifiziert worden.

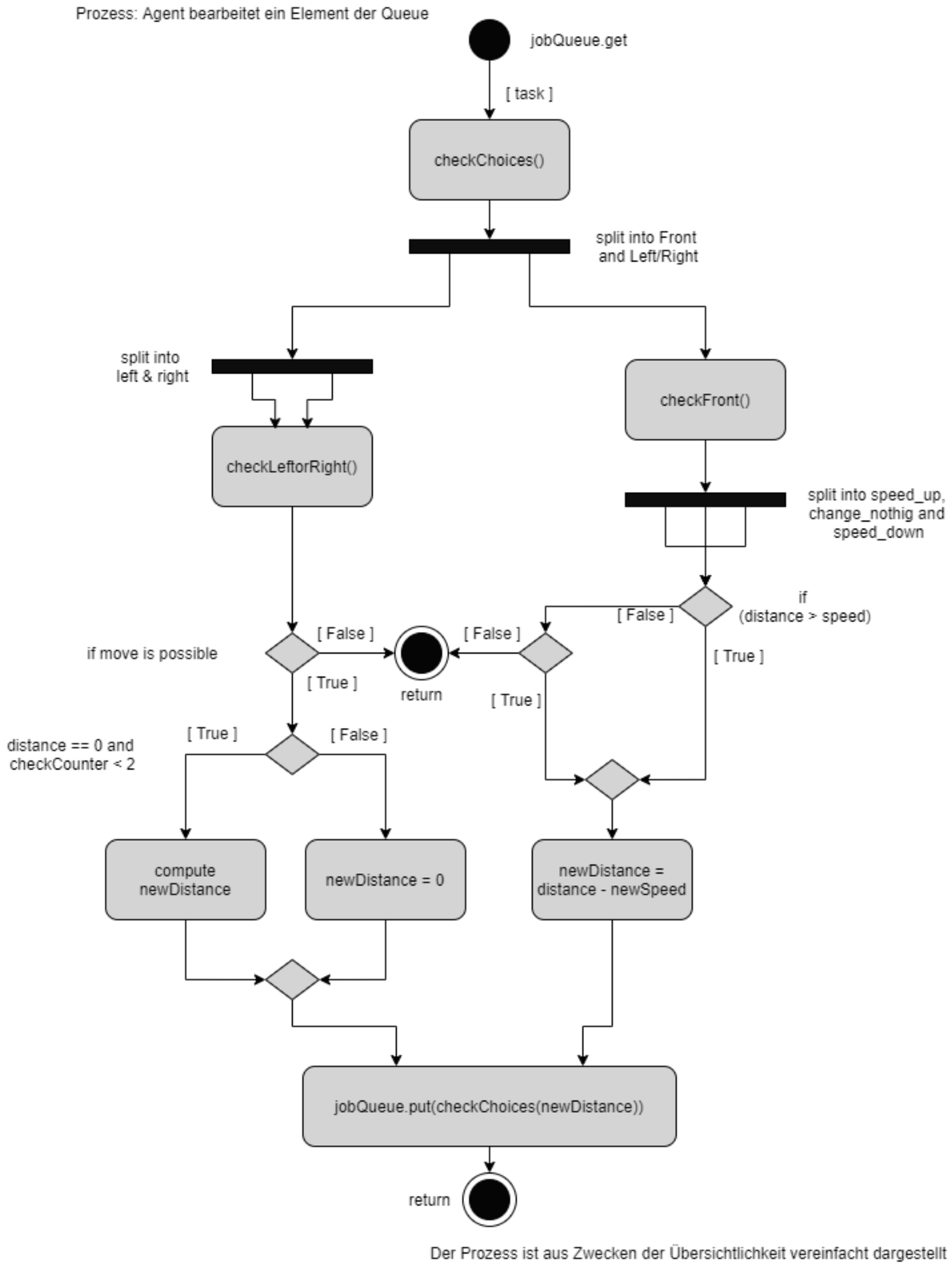


Abbildung 1: Prozess jobQueue.get

2.2 Einzelne Methoden und Variablen-Erklärung

Zuerst werden die Variablen erklärt, damit deren Funktionsweise in den danach erklärten Methoden klar ist.

2.2.1 coord

Die Variable `coord` ist eine Liste von X- und Y-Koordinaten. Wenn die Schlange rekursiv ihre Möglichkeiten testet, muss sie wissen, wo sie schon gewesen ist, um Züge auszuschließen, die in ihr selbst enden. Daher werden diese für jeden Zug jeweils in `coord` gespeichert.

Anfänglich haben wir mit dem Spielbrett gearbeitet und dort die Positionen eingetragen. Da für jeden Zug ein anderes Brett erstellt werden muss, muss dieses kopiert werden. Eine erste Fehlerquelle wurde, dass das Brett mit `newBoard = board` kopiert werden sollte. Jedoch verweist Python bei 2D-Arrays nur auf das ursprüngliche Objekt und macht dort diese Veränderungen auch. Kopierfunktionen wie `board[:]` und `copy(board)` scheiterten aus demselben Grunde. Die erste Rettung war `deepcopy()`, die sich als sehr zeitintensiv herausstellte. Obwohl wir noch eine effizientere Kopiermethode fanden, stellte sich die Koordinatenliste dennoch als effizienter dar. Eine solche Effizienz ist wie schon erwähnt bei so vielen Aufrufen sehr wichtig. Weitere Effizienz erlangte die Liste durch die nachstehenden Optimierungsvariablen `checkCounter` bzw. `coordIgnore`.

2.2.2 checkCounter/coordIgnore

`CheckCounter` wird verwendet, um die Koordinatenprüfung überflüssig zu machen. Dies geschieht, indem beim rechts oder links abbiegen hochgezählt wird und beim Geradeausgehen nur weitergegeben wird. Wenn der Wert kleiner als drei ist, dann kann er sich beim Geradeausgehen nicht in Felder bewegen, welche er in den aktuellen Berechnungen bereits belegt hat und deshalb muss die Koordinatenliste nicht geprüft werden. Wenn der Wert kleiner als zwei ist, ist es auch nicht möglich beim Abbiegen in sich selbst zu lenken.
Eine Optimierung, welche hier beachtet wird, ist, dass der Counter für den Fall, dass eine Richtungsänderung direkt auf eine Richtungsänderung in die jeweils andere Richtung folgt, nicht hochgezählt werden muss. Somit würde der Wert des `checkCounters` für den Fall, dass `turn_left` direkt auf ein initiales `turn_right` folgt, lediglich 1 betragen.
Wenn `checkCounter` echt kleiner als 3 ist, dann wird `coordIgnore` in `checkFront(,` `wenn`es echt kleiner als 2 ist, dann wird `coordIgnore` bei `checkLeftorRight()` auf `True` gesetzt.
`CoordIgnore` sorgt dafür, dass im wahren Fall die Koordinatenliste nicht geprüft werden muss.

2.2.3 collCounter

Der `collCounter` wird verwendet, um die Prüfung von links/rechts zu beschleunigen, indem dieser zählt, wie oft hintereinander in eine Richtung gelenkt wurde. Denn wenn drei Mal in Folge links **oder** drei Mal in Folge rechts gelenkt wurde, trifft die Schlange sich selbst. Prüfungen, ob das Spielfeld verlassen wurde oder ähnliches werden dadurch überflüssig.

2.2.4 logActionValue und Gamma

Die Variable `logActionValue` ist ein 2D-Array, der die "Güte" eines Zuges speichern soll. Initialisiert wird es als leere Liste und wird pro Ebene mit [0, 0, 0, 0, 0] angehängt. Jeder Index der inneren Liste steht dabei für eine Aktion, und zwar: [`speed_up`, `slow_down`, `change_nothing`, `turn_left`, `turn_right`].
Am Beginn einer Runde wird also für die erste Ebene die äußere Liste mit einer Liste von fünf Nullen gefüllt. Jede Möglichkeit wird geprüft und bei Erfolg wird ein Wert ihrem respektiven „Slot" hinzugefügt. Wurden alle geprüft, beginnt die zweite Ebene mit einer weiteren inneren Liste von fünf Nullen. Jetzt werden auf allen erfolgreichen Köpfen die fünf Möglichkeiten geprüft und bei Erfolg ein Wert in dem ursprünglichen Index der zweiten Ebene hinzugefügt. Also für den Fall, dass auf erster Ebene links möglich ist und auf zweite Ebene alle fünf, dann werden die Werte der fünf Züge dem Index von links zugeordnet, weil diese durch das Links-gehen ermöglicht wurden.
Als Wert wird Anfangs eine 1 genommen und wird pro Ebene mit `Gamma` multipliziert, welches standardmäßig auf 0.5 steht, aber verändert werden kann. Ein Wert von 0.5 bedeutet, dass die Werte pro Ebene halbiert werden. Trotz der Halbierung steigen die möglichen Werte pro Ebene, da pro fünf Möglichkeiten nur halbiert wird. Der maximal mögliche Wert pro Ebene errechnet sie durch $x * \left(\frac{5}{2}\right)^t$ wobei x der Anfangswert ist und mit 1 initialisiert wurde und t die Tiefe beschreibt.

2.2.5 calcAction()

Die Methode `calcAction()` rechnet das 2D-Array `logActionValue` zu einer einzelnen Liste zusammen. Dabei wird geprüft, ob alle Züge die gleiche Tiefe erreichen. Ein Zug, der es nicht schafft, bedeutet, dass die Schlange mit diesem Zug in den sicheren Tod steuern würde und wird daher auf -1 gesetzt.

2.2.6 constructEnemyBoard()

Die Methode `constructEnemyBoard()` wandelt das aktuelle Bord so ab, dass alle möglichen Schritte der Gegner mit eingezeichnet werden. Dabei wird nicht berücksichtigt, ob der Gegner bei der Aktion ausscheidet, indem er gegen einen anderen Spieler stößt oder außerhalb des Spielfeldes gerät.
Für jeden Gegenspieler werden alle Aktionen eingezeichnet, wobei berücksichtigt werden muss, ob ein Sprung bevorsteht oder nicht.
Im nicht Sprungfall wird zum Einzeichnen der ersten drei Aktionen: `speed_up`, `change_nothing` und `slow_down` eine Liste erstellt mit der Länge von der derzeitigen Geschwindigkeit des Spielers + 1. Im Falle der maximalen Geschwindigkeit wird die Geschwindigkeit nicht erhöht und eine Variable `speedNotTen` auf 0 gesetzt, damit eine nicht Erhöhung dokumentiert wird. Die komplette Länge der Liste wird in das Bord eingetragen, es sei denn das Spielfeldende wird erreicht. In dem Fall hört es am letzten Feld auf einzutragen.
Für die `turn_left`, `turn_right` Aktionen wird eine Liste erstellt mit einer Länge von der geänderten Geschwindigkeit, wobei -1 gerechnet wird in dem Fall, dass die

Anfangsgeschwindigkeit nicht 10 war. Diese Liste wird, wie bei den ersten drei Aktionen, in das Brett eingetragen.
Im Falle eines Sprungs wird so vorgegangen, dass die Liste wie im nicht Sprungfall erstellt wird, wobei diesmal die Felder mit 0 in die Liste eingetragen werden, die in jedem Fall vom Gegenspieler übersprungen werden.
Die Richtungen, die für einen Gegenspieler eingetragen werden müssen, werden dadurch ermittelt, dass der Vektor zum neuen Kopf berechnet und auf eine Länge von eins vereinheitlicht wird. Damit sind die Summanden bekannt, die auf die aktuelle Position des Gegenspielers addiert werden müssen, wenn er einen Schritt aus der erstellten Liste entnimmt.
Die Methode gibt das neu erstellte Brett zurück.

2.2.7 checkLeftorRight()

Die Methode bekommt als Parameter die Position des vorangegangenen Schrittes, inklusive der Geschwindigkeit, Richtung und Aktion, auf die dieser Zug basiert, übergeben. Zusätzlich noch die Koordinatenliste, mit allen Koordinaten, die die Schlange belegt hat bis zu diesem Schritt, sowie `collCounter`, `checkCounter`, `change` und `distance`. `Change` sagt aus, ob ausgehend von der aktuellen Richtung nach links oder rechts geprüft werden soll und `distance` gibt die Anzahl freier Felder in die Richtung an. Anhand der aktuellen Richtung und der Veränderung wird die neue Richtung berechnet. Zudem wird initialisiert, ob es sich in dieser Runde um einen Sprung handelt.
Als erstes wird der neue Kopf geprüft, ob dieser das Spielfeld verlässt, eine Schlange trifft oder auf eine Koordinate trifft, an der er schon war.
Anschließend wird der Körper der Schlange auf Treffer untersucht. Hier muss nicht mehr geprüft werden, ob diese das Spielfeld verlassen, da dies durch die Prüfung des Kopfes sichergestellt ist. Schwanzaufwärts wird für die Schlange in einer for-Schleife geprüft, ob auf dem Weg eine andere war oder die Koordinaten in der Liste sind (vgl. `coordIgnore`). Handelt es sich um eine Sprung-Runde, brauch nur der Schwanz getestet werden. Wurde die gesamte Schlange fehlerfrei überprüft, wird der entsprechenden Aktion in `logActionValue` ein Wert für diese Tiefe zuaddiert. Abschließend wird ausgehend auf diesem Kopf und Richtung durch `checkChoices()` eine Prüfung aller fünf Möglichkeiten am Ende der Warteschlange gestellt.
Für den Fall, dass die Distanz, welche mit `checkChoices()` in die Warteschlange übergeben würde, 0 betragen würde und `coordIgnore` `True` ist, wird mit Hilfe von `checkDistance()` die neue Distanz berechnet und danach übergeben. Dies erhöht die Effizienz des Codes, da durch eine übergebene Distanz potenziell weniger Prüfungen bei `checkChoices()` getätigt werden müssen.

2.2.8 checkFront()

`CheckFront()` überprüft die Zugoptionen, die der Spieler hat, wenn er geradeaus geht. Eine Option steht dann zur Verfügung, wenn er durch eine Aktion nicht direkt ausscheidet. Dabei werden die Aktionen `change_nothing`, `speed_up` und `slow_down` in einem Aufruf der Methode kontrolliert. Dabei gibt es einen Effizienz Vorteil, da alle drei Aktionen sich Felder teilen, die nur einmal überprüft werden müssen.

Die Methode bekommt Argumente, die den Zustand des Spielers beschreiben, wie `direction`, `x` Koordinate, `y` Koordinate und die Geschwindigkeit. `Depth` und `InitialAction` sind zwei Argumente, die übergeben werden, damit die richtige Position im `logActionValue` 2D Array verändert werden kann. Zudem wird noch die Koordinatenliste `coord` bereitgestellt, aus Gründen, die unter 2.2.1 genannt wurden.
Das Argument `distance` wird dafür verwendet, die Feldprüfungen zu vereinfachen. Der Agent überprüft einmalig wie viel Platz er nach vorne hat, bis Spielfeldende oder ein belegtes Feld erreicht wird. Dadurch können bei wiederholten geradeaus Schritten die Überprüfung nur anhand der freien Felder bestimmt werden. Als letztes Argument wird der `checkCounter` übergeben, den wir bereits erläutert haben.
Im ersten Schritt wird der `stepVector` bestimmt. Der gibt an, welche Summanden für den Schritt der Länge eins auf die Koordinaten addiert werden müssen, damit der Spieler sich weiter in seine derzeitige Richtung bewegen kann. Damit kann der neue Kopf bei den zu prüfenden Aktionen berechnet werden. Um die Möglichkeit der drei Aktionen zu speichern wird ein Array erstellt, welches mit drei `False` Einträgen initialisiert wird. Sobald es überprüft wurde, ob im Sprungfall die Aktion möglich ist, wird der zugehörige Array Eintrag der Aktion auf `True` gesetzt. Diese Möglichkeit wird durch mehrere Prüfungen ermittelt wobei drauf geachtet wird, dass so wenig Abfragen wie möglich gestellt werden. Dafür wird auch das Argument `distance` genutzt. Wenn die aktuelle Geschwindigkeit kleiner oder gleich der übergebenen Distanz ist, werden die Kopfprüfungen weitgehend umgangen.
Nachdem die Köpfe geprüft wurden, wird noch kontrolliert ob in dem nicht Sprungfall ein Feld zwischen altem und neuem Kopf belegt ist. In dem Fall wird die Methode abgebrochen und kein weiterer Job in die Queue gesetzt.
Nach der Kontrolle wird für jede Option, die der Agent hat, der zugehörige `logActionValue` um den diskontierten Wert `self.Value` erhöht. Zusätzlich wird geprüft, ob der Spieler sich im Deadend befindet und ob er daraus springen kann, wenn er die Aktionsoption wählt. In dem Fall wird ein deadendBias addiert, der standardmäßig 500 beträgt.
Für alle Optionen wird ein neuer Job in die Queue gelegt. Der beinhaltet die Position, Richtung und Geschwindigkeit, die er nach dem Wählen der Aktion haben würde.

2.2.9 checkChoices()

Beim Aufruf von `checkChoices()` wird als erstes geprüft, ob noch genügend Zeit bis zur Deadline verbleibt. Sollte dies der Fall sein, wird geprüft, ob dies der erste Aufruf auf neuer Ebene ist. Falls ja, wird die `logActionValue` erweitert, sowie rundenbasierte Variablen (Wert der Ebene und Rundencounter) aktualisiert. Dann werden nur noch `checkFront()`, und `checkLeftorRight()` mit links und rechts aufgerufen.

2.2.10 jobQueue

Die `jobQueue` sorgt dafür, dass die Prüfungen Ebene für Ebene ablaufen. Angefangen hatten wir mit Rekursion und return-Statements, jedoch war der Ansatz nicht zeitflexibel, da eine vorher festgelegte Tiefe bestimmt werden musste. Ohne die Tiefe und nur ein zeitliches Abbruchkriterium rechnete die DommeAI die erste Aktion in die tiefst mögliche Ebene durch bis keine Zeit mehr blieb.

2.3 Strategien

Um die Spielweise der DommeAI weiter zu verbessern, haben wir neben der Grundstrategie zwei weitere Spielstrategien implementiert, welche die Spielweise in bestimmten Situationen verändern. Diese bestimmten Situationen werden unter anderem durch den Wert der `checkDeadend()` Methode, so wie der `getMinimalEnemyDistance()` Methode, erkannt, welche im Folgenden erklärt werden. Nach den Methoden werden die Variablen erklärt, welche einen Einfluss auf die Strategien und ihre Berechnungen haben.
Die verschiedenen Strategien und deren Erkennung werden in einem darauffolgenden Schaubild dargestellt. Bei den zwei zusätzlichen Strategien handelt es sich um eine Strategie, welche das Verhalten steuert, falls sich der Spieler in einer mutmaßlichen Sackgasse befindet und um eine Strategie, nach welcher operiert wird, wenn sich DommeAI in einer safeZone befindet. Diese Strategien werden abschließend ebenfalls im Detail erläutert.

2.3.1 checkDeadend()

In der `checkDeadend()` Methode wird ein Wert berechnet, welcher ausgehend von dem aktuellen Spielbrett und den gegebenen Koordinaten ein Maß angibt, welches das Ausmaß des freien Platzes um die übergebenen Koordinaten beschreibt. Je kleiner dieser Wert ist, desto eher befinden sich die gegebenen Koordinaten in einer Sackgasse. Neben der Erkennung von Sackgassen, wird dieser Wert auch bei der Erkennung von safeZones benutzt. Eine safeZone liegt dann vor, wenn der Wert von `checkDeadend()` in der aktuellen Situation über der Variable `safeZoneLimit` (35) liegt und die Manhattan Distanz zum nächsten Gegner mindestens dem Wert von `safeZoneDistance` (20) beträgt (vgl. `getMinimalEnemyDistance()`).

Um die Berechnungen bei `checkDeadend()` zu verdeutlichen, werde ich sie am folgenden Beispiel visualisieren:

Der Spieler, welcher durch eine grüne Farbe und einem grauen Kopf dargestellt wird, befindet sich in einer geschlossenen Zone.

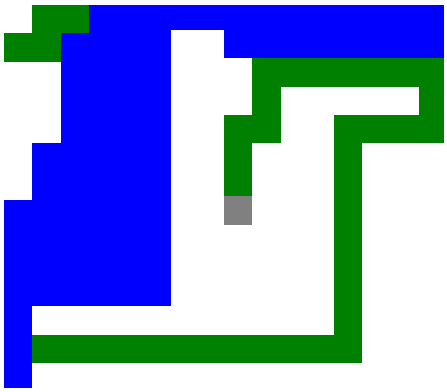
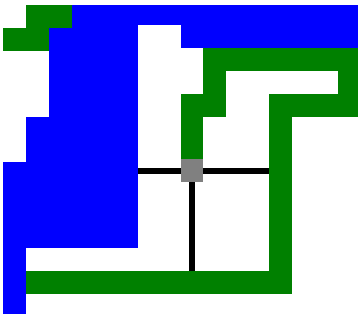


Abbildung 2: Ausgangssituation checkDeadend()



Zuerst wird die Distanz in jeder der möglichen Richtungen erfasst (links, vorne, rechts)

Abbildung 3: checkDeadend() Teil 1

Daraufhin wird in die erste Richtung (links von der Orientierung der Schlange) berechnet. Das Ergebnis ist die maximale T-Distanz in der Richtung, also die Distanz in der Richtung addiert mit der maximalen Distanz von dem letzten freien Feld ausgehen nach links und rechts.

In diesem Fall beträgt die maximale Distanz in der Richtung 3 Felder.
Davon ausgehend beträgt die maximale Distanz nach links oder rechts 4 weitere Felder. Die maximale Distanz nach rechts oder links wird durch die Methode `maxLR()` bestimmt.
Somit ergibt sich in dieser Richtung der Wert 7 (= 3+4).

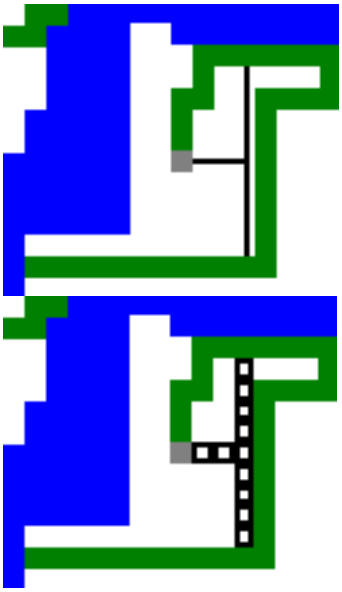


Abbildung 4: checkDeadend() Teil 2+3

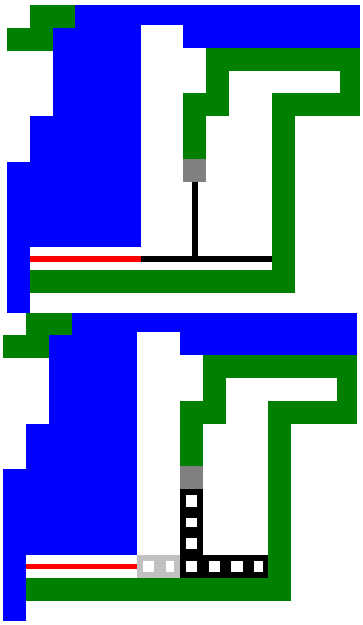


Abbildung 5: checkDeadend() Teil 4+5

Die nächste zu überprüfende Richtung ist die der Schlange.
In diesem Fall tritt bereits ein Sonderfall auf, welcher abgefangen wird. So wird die Distanz nach vorne und dann nach rechts der Schlange nicht komplett gewertet, da es sich um eine Spalte handelt, welche lediglich eine Breite von 1 hat. Somit könnte die Schlange dort (ausser durch Sprünge) keine weiteren Züge machen. Deshalb werden lediglich die Distanzen gewertet, in welchen der Schlange mindestens ein freies Feld am Ende der Distanz zur Verfügung steht, was durch die Methode `freeLR()` ermittelt wird, welche innerhalb von `maxLR()` aufgerufen wird. Somit beträgt der Wert nach vorne lediglich 7. Dieser setzt sich aus der maximalen Distanz nach vorne (4) und der maximalen Distanz links und rechts des letzten freien Feldes (3) zusammen. Die Distanz nach vorne und dann nach rechts würde lediglich 6 betragen (4+2).

Der letzte und größte Wert aus der gegebenen Situation ergibt sich aus der letzten zu überprüfenden Richtung (rechts der Schlange). Dort ist der Wert 8, welcher sich aus der Distanz rechts der Schlange (2) und der maximalen Distanz nach links / rechts (6) ergibt.
Somit berechnet die Methode `checkDeadend()` in der gegebenen Situation den Wert 8, welches dem Maximum der drei Werte entspricht.

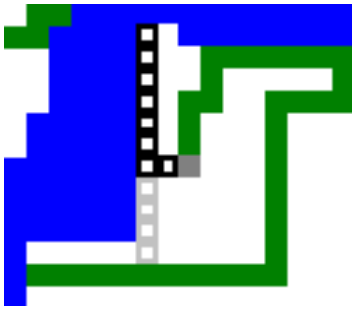


Abbildung 6: checkDeadend() Teil 6

Ein weiterer Sonderfall ergibt sich in der Folgenden Situation:

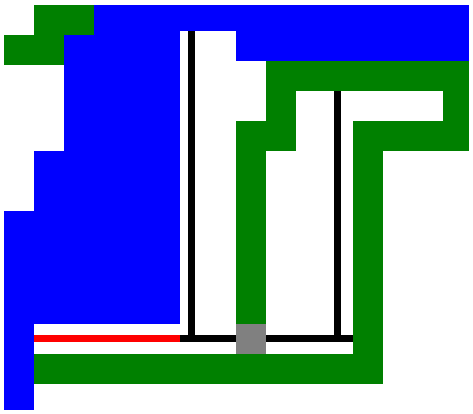


Abbildung 7: checkDeadend() Teil 7

Dort ist zu sehen, dass die Berechnung in der Richtung rechts der Ausrichtung der Schlange nicht von dem letzten freien Feld ausgeht, sondern von dem letzten freien Feld, in welchem nach links oder rechts mindestens ein freies Feld verfügbar ist. Dieser Sonderfall wird in der Methode `checkDeadend()` abgefangen, welche erfasst, wenn `maxLR()` für ein bestimmtes Feld 0 zurückgibt und in dem Fall das vorherige Feld überprüft.

Somit gibt die Methode in dieser Situation den Wert 12 wieder, welcher dem Wert rechts der Schlange entspricht (2+10). Dieser Wert wird gewählt, da die Werte nach vorne (0) und nach links der Schlange (3+8) kleiner als 12 sind.

Aus Optimierunsggründen bricht die Berechnung in `checkDeadend()` ab, sobald in einer Richtung ein Wert berechnet wurde, welcher größer ist als das gewählte `safeZoneLimit`, ab welcher eine potenzielle safeZone vorliegen könnte (35). Für den Fall, dass mit der Methode lediglich berechnet werden soll, ob es sich bei einer Situation um eine Sackgasse handelt, bricht er schon ab Werten ab, welche über dem `deadendLimit` (14) liegen.

2.3.2 getMinimalEnemyDistance()

In der Methode `getMinimalEnemyDistance()` wird auf Basis der Manhattan Distanz die Distanz zu dem aktiven Gegner berechnet, welcher am nächsten am Spieler ist.
Die Manhattan Distanz beschreibt die Distanz zwischen zwei Punkten, welche entlang der Achsen in rechten Winkeln gemessen wird. In diesem Fall wird die Manhattan Distanz beispielsweise zwischen (x1, y1) und (x2, y2) berechnet als |x1 – x2| + |y1 – y2| (Paul E. Black 2019).
Der Methode wird der aktuelle `state` übergeben, so wie die Koordinaten des Spielers. Sie gibt die Manhattan Distanz zum nächsten Gegner als `Integer` zurück.

2.3.3 overSnake()

In der `overSnake()` Methode wird geprüft, ob der Spieler im Zuge eines potenziellen Sprunges über andere Schlangen, beziehungsweise seinen eigenen Körper, springen würde.
Ihr werden die Koordinaten übergeben, von welcher ein potenzieller Sprung ausgehen würde, so wie das Spielfeld, die Ausrichtung der Schlange und die Geschwindigkeit der Schlange. Dann wird in der Methode bestimmt, ob die freien Felder, welche durch den Sprung entstehen würden, von Spielern belegt wären oder frei wären. Wären diese Felder alle frei, so gibt diese Methode `False` wieder, während sie `True` wiedergibt, falls mindestens eine Schlange übersprungen werden würde.

2.3.4 Erläuterung der Strategievariablen

`deadendLimit`	Das `deadendLimit` beschreibt die Grenze, unter welcher der Wert von `checkDeadend()` als Sackgasse interpretiert wird
`isDeadend`	`isDeadend` ist `True`, wenn der Wert von `checkDeadend()` einer Runde < `deadendLimit`; wird am Anfang einer Runde für diese festgelegt
`deadendBias`	Beschreibt den Wert, welcher im Falle von `isDeadend` = `True` dem nächsten Zug addiert wird, welcher potenziell zu einem Sprung in einen Raum führen könnte, welcher keine Sackgasse ist
`safeZoneLimit`	Das `safeZoneLimit` beschreibt die Grenze, über welcher der Wert von `checkDeadend()` als potenzielle safeZone interpretiert wird
`safeZone-Distance`	Die `safeZoneDistance` beschreibt die Grenze, über welcher der Wert von `getMinimalEnemyDistance()` als potenzielle safeZone interpretiert wird
`isSafeZone`	`isSafeZone` ist `True`, wenn der Wert von `checkDeadend()` > `safeZoneLimit` und der Wert von `getMinimalEnemyDistance()` > `safeZoneDistance` Dann wird der aktuelle Stand des Spielers als safeZone gewertet
`safeZoneBias`	Beschreibt den prozentualen Anteil, um den im Falle einer safeZone der Wert von `speed_up` gemindert wird und um den der Wert von `slow_down` gesteigert wird

Tabelle 1: Erläuterung der für Strategien relevanten Variablen

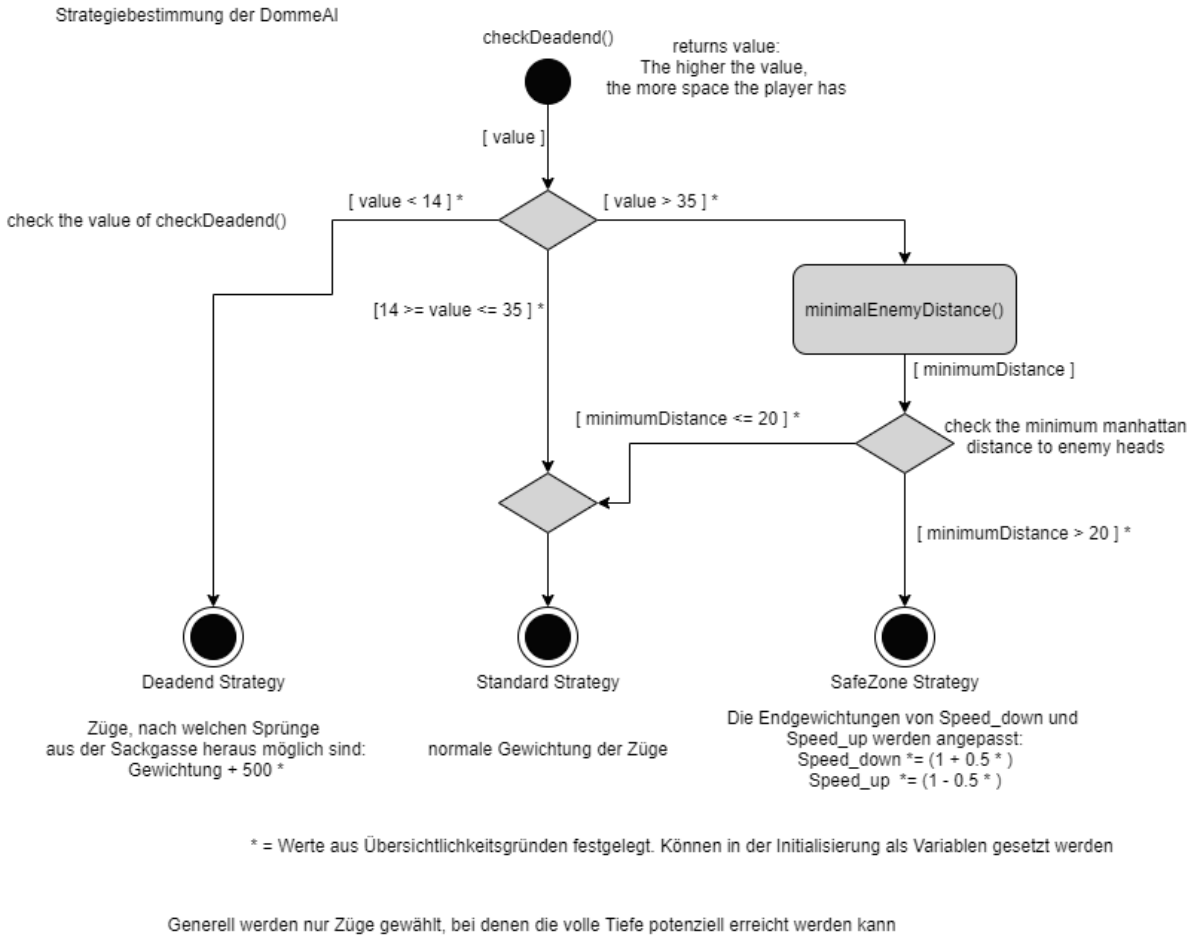


Abbildung 8: Strategiebestimmung

2.3.5 Deadend Strategie

Falls der Wert von `checkDeadend()`, welcher am Anfang der Berechnungen ermittelt wird, unter dem gewählten `deadendLimit` liegt (14), operiert die DommeAI nach der Deadend Strategie (vgl. Abbildung 8). Somit wurde erkannt, dass eine Sackgasse vorliegt und es wird diese Strategie gewählt, damit der Spieler aus der Sackgasse springt.
In dieser Strategie wird bei jedem möglichen Sprung ermittelt, ob der Spieler in diesem Sprung über andere Schlangen springen würde (`overSnake()`) und ob der Spieler nach dem Sprung noch in einer Sackgasse wäre (`checkDeadend()`). Falls der Sprung über eine Schlange erfolgen würde und der Spieler sich nach dem Sprung nicht mehr in einer Sackgasse befinden würde, wird der initiale Zug, welcher zu diesem Sprung geführt hat, mit dem `deadendBias` bevorzugt. Dieser Bias führt dazu, dass die DommeAI in dem Fall, dass sie in einer Sackgasse ist und einen möglichen Sprung aus dieser Sackgasse ermittelt hat, einen Zug macht, welcher zu einem solchen Sprung führen könnte, falls sie trotzdem die maximale Tiefe der Züge für diesen Zug berechnet hat.
Dieser Bias wird unabhängig davon vergeben, in welchem der nächsten Züge der Sprung erfolgen würde. Falls also beispielsweise ein initialer Zug nach links zur Folge hätte, dass ein Sprung in 9 Zügen möglich wäre, während ein Zug nach rechts zur Folge hätte, dass ein Sprung

nach 3 Zügen möglich wäre, würden sowohl der initiale Zug nach rechts als auch der nach links den gleichen Bias des `deadendBias` erhalten.
Diese Strategie kann beispielsweise auf Abbildung 11 im Anhang beobachtet werden. So ist zu sehen, dass `isDeadend` (im GUI mit „DE“ abgekürzt) `True` entspricht. Somit werden Züge höher bewertet, welche zu einem potenziellen Sprung aus der Sackgasse führen könnten. In der darauffolgenden Abbildung (Abbildung 12 im Anhang) ist dann zu sehen, dass der Sprung erfolgt ist und der Wert für `isDeadend` wieder auf `False` gesetzt wurde.

2.3.6 SafeZone Strategie

Die safeZone Strategie wird gewählt, falls die Manhattan Distanz zu dem nächsten aktiven Gegner (`getMinimalEnemyDistance()`) über der gewählten `safeZoneDistance` (20) liegt und der Wert von `checkDeadend()` über dem gewählten `safeZoneLimit` (35) liegt (vgl. Abbildung 8). Demnach wurde erkannt, dass sich der Spieler in einem großen freien Areal befindet und aufgrund der hohen Distanz zu dem nächsten aktiven Gegner innerhalb der nächsten Züge nicht durch Züge des Gegners gefährdet werden kann.
Falls dies der Fall ist, soll eine tendenziell langsamere Geschwindigkeit bevorzugt werden, damit die DommeAI möglichst lange überlebt, ohne zusätzliche Risiken einzugehen. Deshalb wird der initiale Zug `slow_down` um den gewählten `safeZoneBias` (50%) höher bewertet, während `speed_up` um den gewählten `safeZoneBias` niedriger bewertet wird.
Dieser Bias sollte nicht zu hoch gewählt werden, damit die Zugentscheidungen weiterhin maßgeblich durch die Grundstrategie geprägt werden und lediglich eine Tendenz zu geringeren Geschwindigkeiten vorliegt.
Diese Strategie ist auf Abbildung 13 im Anhang zu sehen. Da der Wert von `checkDeadend()` (im GUI mit „de“ abgekürzt) über dem gewählten safeZoneLimit liegt, und die Distanz zu den Gegnern über der `safeZoneDistance` liegt, wird die safeZone Strategie gewählt. Somit wird unter anderem durch den Bias, welcher die Werte von `slow_down` und `speed_up` beeinflusst, die Entscheidungsmöglichkeit `slow_down` ausgewählt.

3 Theoretische Ausarbeitung

3.1 Hintergrund

Der generelle Ansatz der Lösung ist es, die verschiedenen Zugmöglichkeiten in steigender Tiefe rekursiv mit Hilfe einer Queue zu testen. Dieser Ansatz erlaubt dem Programm eine hohe Flexibilität in vielen Hinsichten. Somit verwenden wir den Ansatz des Branch and Bounds (Prof. Dr. Marco Lübbecke 2018).
Dieser Ansatz ermöglicht eine hohe Anpassungsfähigkeit an verschiedene Unsicherheiten. So sind beispielsweise Differenzen in Aspekten wie der Zeit, welche dem Programm pro Zug zur Verfügung steht, der Rechenleistung oder der Eckdaten des Spieles zu beachten. Da es ein Teil der Aufgabenstellung ist, mit diesen Unsicherheiten umzugehen, eignet sich unser gewählter Ansatz besonders gut.

3.2 Auswertung & Diskussion

Die Qualität der Lösung lässt sich schwer quantifizieren, da sie in Relation zu anderen Lösungen gesehen werden muss. Wir konnten viele Testspiele spielen und dort feststellen, dass sie sich ziemlich gut schlägt und viele Situationen gut absolviert. Dabei können hier keine verlässlichen Sieg Statistiken genannt werden, da verschiedene Bots in den Spielen waren, die keine wirklichen Abgaben darstellen.
Die Güte unserer Lösung ist primär von der zur Verfügung stehenden Rechenleistung und Zeit abhängig. Jedoch begrenzt das exponentielle Wachstum stark die Tiefe der Berechnungen. Selbst wenn die Rechenleistung, sowie Zeit verdoppelt werden, kann es sein, dass nicht einmal eine Ebene mehr zu Ende geprüft werden kann.
Die Entscheidungen, die die Schlange trifft, sind jedoch in jedem Fall nachvollziehbar. Die Schlange strebt danach möglichst viele Züge in Zukunft machen zu können. Dies führt unter anderem dazu, dass die Geschwindigkeit im mittleren Bereich anvisiert wird, da diese die meisten Möglichkeiten von `speed_up` und `slow_down` bieten. Dieser „Branch-and-Bound“-Ansatz ist in allen Situationen sehr flexibel, da es jede Möglichkeit prüft. Obwohl unsere Lösung von viel Zeit und Rechenleistung profitiert, ist sie doch auch sehr vorausschauend, wenn von beiden nicht so viel bereitsteht. Denn die ersten Ebenen kann er schon mit wenig Leistung innerhalben von Bruchteilen einer Sekunde errechnen. Außerdem ist der Ansatz sehr flexibel bezüglich Spieleranzahl und Feldgröße und die erwähnten Strategien mit ihren Gewichtungen können angepasst werden.

4 Softwarearchitektur und -qualität

4.1 Software testing

Die Software wurden in PyCharm als auch im Docker ausgiebig auf Reliabilität und Validität getestet, um Fehler zu finden und zu eliminieren. Doch wie Dijkstra richtig feststellte, kann man „durch Testen […] stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen.“ (Edsger W. Dijkstra 1972) Daher wollen wir hier keine Fehlerfreiheit garantieren, sind uns durch das viele Testen jedoch ziemlich sicher, dass keine größeren Fehler enthalten sind. Die Tests konnten wir gut durch das GUI nachvollziehen, um anhand der angezeigten Werte potenzielle Fehler zu finden. Fehlerhafte `states` wurden lokal intensiv getestet, um den Fehler einzugrenzen und nach Behebung zu überprüfen, ob dieser noch in der Situation auftaucht.

4.2 Coding conventions

Für die Programmierung haben wir uns an allgemein üblichen Konventionen gehalten. Variablen, Methoden, so wie Kommentare wurden auf Englisch verfasst, um auch internationalen Nutzern des Codes das Verständnis zu vereinfachen. Dabei wurden Namen im „camel case“ geschrieben, wie es in vielen Programmiersprachen üblich ist, um die Namensgebung übersichtlicher zu gestalten. Ferner wurden logisch zusammenhängende Teile innerhalb der Methoden von anderen durch Absätze getrennt und mit Kommentaren erläutert. Die Verwendung von „docstrings“ am Anfang einer Methode gibt einen groben Überblick über die Methode und ihre Parameter und ist in der Programmierung Standard. Außerdem wurden optische

Konventionen von Python, wie zwei Zeilen zwischen zwei Methoden oder die Begrenzung der Zeilenlänge, eingehalten, um die Lesbarkeit zu gewährleisten.

4.3 Wartbarkeit

Das System kann mit kleinem Aufwand gewartet bzw. angepasst werden. Wichtige Werte an denen geschraubt werden kann (z. B. Bias oder ähnliche) werden in der `__init__()` initialisiert und sind daher leicht änderbar. Die Klassen wurden möglichst modular aufgebaut, um spätere Änderungen zuzulassen (vgl. Anhang: Abbildung 10, Klassendiagramm DommeAI).

5 Zusätzliche Abgaben

5.1 GUI

Während der Entwicklung der DommeAI hat sich organisch ergeben, dass ein GUI, welches das Spiel live anzeigen würde, in vielen Aspekten hilfreich wären. Die primären Gründe waren das Erkennen von Schwachstellen und das Debugging. Durch die visuelle Analyse der Spielweise konnten verschiedene Strategie- und Optimierungsansätze entdeckt werden, die schon beschrieben wurden. Leider funktioniert die Programmierung nicht von Anfang an fehlerfrei und daher war es enorm hilfreich Fehlentscheidungen der DommeAI visualisiert zu bekommen, um den Fehler im Code eingrenzen zu können.
Somit haben wir es so umgesetzt, dass wir mit Hilfe der matplotlib-Bibliothek für Python nach jedem Zug der DommeAI die Matrix des Spielfeldes, inklusive vieler Metainformationen, mit welcher die Entscheidungen im Programm getroffen werden, visualisiert werden. Es wird also nach jedem Zug eine .jpg Datei erstellt, mit welcher man das Spiel live verfolgen kann. Insbesondere in Kombination mit dem SciView-Feature (jetbrains.com 2021) von PyCharm Professional, welches für Studenten kostenlos ist, eignet sich dieses GUI sowohl zum Debuggen als auch zum Anschauen und verfolgen des Spielverlaufes sowohl während des Spiels, als auch im Nachhinein (vgl. Anhang, Abbildung 12).
Auch bei Verwendung eines Dockers kann der Spielverlauf im Nachhinein betrachtet werden, indem die gespeicherten Bilder aus dem Docker extrahiert werden. Eine kurze Anleitung dafür ist im Handbuch enthalten.
Als kleines Gimmick haben wir das GUI durch die Bereitstellung durch matplotlib im Stil der xkcd-Comics (Randall Munroe 2021) erstellen lassen. Dies könnte im Code mit geringem Aufwand dazu geändert werden, dass ein normaler Stil genutzt wird.

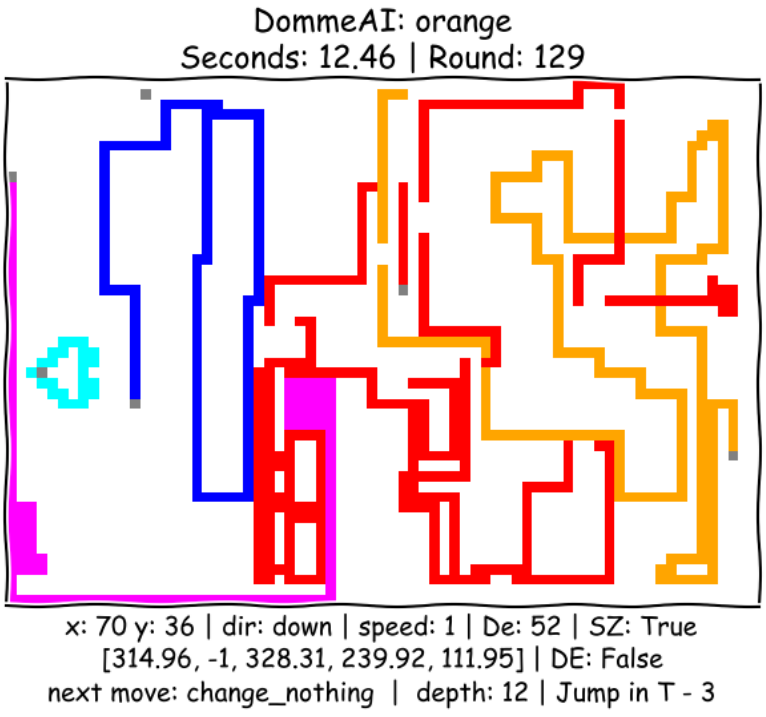


Abbildung 9: Beispielansicht eines Spiels

5.2 TensorFlow

Machine Learning wird dafür verwendet auf Grundlage von Erfahrungswerten Schätzungen zu berechnen. Unsere Idee war es, mit dieser Methode eine Künstliche Intelligenz für Spe_ed zu entwickeln. Zur Umsetzung haben wir uns für das TensorFlow Framework entschieden, da es für unseren Anwendungsfall ausführlich dokumentiert ist.

Die gängigste Methode zum Erlernen eines Sachverhaltes ist das Supervised Learning. Dabei werden vorhandenen Trainingsdaten genutzt, um ein neuronales Netz zu trainieren. Der Datensatz besteht aus Eingabedaten, für die eine Schätzung vorgenommen werden soll, und einer entsprechenden Lösung. Das neuronale Netz berechnet eine Schätzung auf Grundlage der Eingabedaten und vergleicht diese mit der mitgelieferten Lösung. Je nach Abweichungen können die Parameter des neuronalen Netzes angepasst werden, um bessere Schätzungen zu erreichen.

Für den Anwendungsfall Spe_ed, belaufen sich die Eingabedaten auf die Zustände des Spiels. Diese werden vom Server übermittelt und können als Grundlage für die Schätzung genutzt werden. Vor allem das Spielbord und die Daten des Spielers sollten zum Trainieren eines neuronalen Netzes elementar sein.

Im Vergleich zu dem Supervised Learning, gibt es im Anwendungsfall keine mitgelieferten Lösungen für die Eingabedaten. Es ist auch meistens nicht möglich, exakt zu bestimmen, welche Aktion in einem bestimmten Spielzustand optimal ist. Durch den Entfall der Lösungsdaten kann Supervised Learning nicht verwendet werden. Deswegen haben wir einen Reinforcement Learning Ansatz verfolgt. (Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar 2018)

Beim Reinforcement Learning werden keine Trainingsdaten in Form von Zustand - Lösung benötigt. Stattdessen wird mit einem Environment interagiert, welches für eine Aktion einen Reward zurückgibt. Der Reward gibt Auskunft darüber, wie gut eine oder eine Reihe an Aktionen war. Aufgabe des Lernalgorithmus ist es, den Reward über eine gewisse Anzahl an Aktionen zu maximieren. Der Teil des Algorithmus, der mit dem Environment interagiert wird Agent genannt.

Grundlage des Reinforcement Learnings bildet das Environment, welches den Reward zurückgibt, der Aussage darüber gibt, ob eine Aktion in einem gewissen Zustand vom Agenten verstärkt oder bestraft werden soll. Bei Spe_ed wird das Environment vom Spiel gebildet, dass bei einer der fünf möglichen Aktionen einen Reward zurückgibt. Der kann beispielsweise davon abhängen, ob der Spieler nach der Aktion immer noch aktiv ist oder durch die Aktion ausgeschieden ist. Das Environment wird mit der gymAi von OpenAI umgesetzt, welches eine Art Adapter darstellt, mit dem der TensorFlow Agent arbeiten kann.

Da der Agent einen gesamten Reward maximieren möchte, um bei uns beispielsweise so lange wie möglich im Spiel aktiv zu bleiben und nicht nur im nächsten Schritt, wird ein Q-learning

Ansatz verfolgt, der auf der Bellmann Funktion aufbaut. Die sagt aus, dass der Langzeit Reward einer gegebenen Aktion in einem bestimmten Zustand, dem direkten Reward entspricht, der die Aktion vom Environment zurückbekommt plus dem diskontierten maximal zu erwartenden Reward aus dem neuen Zustand, der durch die Aktion erreicht wird:

$$Q(s,a) = r + \gamma(\max(Q(s',a'))$$

Bei der Wahl des maximalen Q-Values, der in einem Zustand erreichbar ist, kann die beste Lösung gewählt werden.
Die Aufgabe des Reinforcement Learning Algorithmus ist es, den Q-Value für alle Aktionen mit einem Zustand als Eingabe zu schätzen. Dafür wird ein neuronales Netz verwendet, dass den Zustand als Eingabe bekommt und fünf Werte ausgibt, jeder repräsentiert den Q-Value für eine der fünf Aktionen. Dieser kann mit dem ausgerechneten Wert der Bellmann Funktion verglichen werden. Dafür wird der erhaltene Reward genutzt und der maximale zukünftige Q-Value, der durch das neuronale Netz geschätzt wird. Durch die Abweichung kann das neuronale Netz angepasst werden. Durch die iterative Anpassung des Netzes, wird der Q-Value exakter geschätzt und somit können bessere Aktionen gewählt werden. (Arthur Juliani 2016)

Das Vorgehen des Reinforcement Learnings wurde vereinfacht dargestellt. Es gibt viele Features, die den Lernprozess verbessern und damit zu kontinuierlich besseren Lösungen führen wie beispielsweise das Einführen eines Target Networks, experienced Replay oder die Nutzung eines Asynchronous Actor-Critic Agents.

Wir haben den Ansatz entwickelt jedoch nach einer gewissen Zeit verworfen, da uns zum einen die Erfahrung mit neuronalen Netzen fehlt, aber auch die Hardware, um das Netz für unseren Anwendungsfall zu optimieren.
Durch die fehlende Erfahrung und die hohe Dichte an Parametern, die in so einem Netz optimiert werden müssen, muss das Netz oft trainiert werden, welches je nach Hardware ein langwieriger Prozess sein kann.

In der Abgabe ist zum einen das nachprogrammierte Spiel zu finden, als auch der Tensorflow Agent. Das Spiel Spe_ed haben wir nachprogrammiert, um schneller Spielzustände zu bekommen. So konnten mehrere tausend Spiele in wenigen Sekunden berechnet werden. Die Lösung ist in einem unfertigen Zustand.

5.3 Trash-Talk

Diese „Erweiterung“ ist als kleiner Spaß beim Debugging entstanden, da manchmal am Ende eines Spiels unklar war, ob das Spiel gewonnen wurde (besonders vor der Entstehung des GUI). Daher haben wir erst uns drucken lassen, ob das Spiel „gewonnen“ oder „verloren“ wurde, je nachdem wie am Ende unser Zustand ist. Aus „gewonnen“ wurden die ersten Nachrichten, die als sogenannter „Trash-Talk“ aus vielen online Mehrspieler-Spielen stammen.

6 Fazit

6.1 Ausblick

Abschließend möchten wir an dieser Stelle einen Ansatz erwähnen, den wir leider aus Zeitgründen nicht umsetzen konnten.
Dieser wäre es, die durchgerechneten Züge zu speichern, um diese im nächsten Zug wieder zu verwenden und an der Stelle an der aufgehört wurde weiter zu machen. Dies kann man sich als Entscheidungsbaum vorstellen, auf dem pro Knoten 5 neue Knoten hinzukommen. Wird sich für ein Schritt entschieden, dann können auf erster Ebene die anderen 4 Knoten samt Nachfolgenden gestrichen werden. Außerdem müssen Knoten die durch einen Zug vom Gegner unmöglich gemacht wurden auch samt Nachfolgenden gestrichen werden. Dieser Ansatz hat uns zwar sehr fasziniert, jedoch war er durch die sich nähernde Abgabefrist ausgeschlossen.
Auf der Kehrseite sollte hier jedoch erwähnt werden, dass zum einen sehr viel gespeichert werden muss und der Zugewinn sehr gering ist. Gespeichert werden müsste für jeden Kopf die Koordinaten, Geschwindigkeit, Richtung und eine Liste, wo die Schlange schon gewesen ist. Bei einer üblichen Tiefe von 8 würde dies bedeuten, dass bis zu $5 * 5^8 = 1\,953\,125$ Elemente gespeichert werden müssten. Eine Ebene Tiefe wären es schon fast 10 Millionen Elemente, wovon ein Element eine Liste ist. Der gegenüberstehende Nutzen ist eher gering, denn wie erwähnt ist es durch das exponentielle Wachstum eine Ebene tiefer schon bis zu fünf Mal so aufwendig. Eine Situation, wo dieser Ansatz jedoch sehr vorteilhaft ist, ist wenn in einer Runde nur wenige Sekunden bereitgestellt werden. Unser aktueller Ansatz könnte erst viel rechnen und einen wichtigen Schritt planen, um z. B. zu springen. Wenn jedoch im nächsten Zug nicht genug Zeit bereitsteht, um diesen Gedanken weiter zu verfolgen, weil die Schlange nicht so tief rechnen konnte, macht die Schlange möglicherweise falsche Züge und sabotiert ihren Gedanken. Hier wäre so eine Optimierung enorm hilfreich, da die Schlange nicht mehr den Gedanken vergisst, sondern weiterverfolgt.

6.2 Schlusswort

Insgesamt war das Projekt für uns als Gruppe eine sehr interessante Aufgabe und eine willkommene Abwechslung zum sonst eher theoretischen Uni-Alltag. Es hat sich zwar besonders im Verlauf des Projektes als viel Arbeit herausgestellt, jedoch hat die Arbeit in der Gruppe an weitestgehend interessanten Problemen auch viel Spaß gemacht.
Man konnte sowohl unterschiedlichste Elemente aus den verschiedenen Vorlesungen des bisherigen Studiums vertiefen und anwenden als auch neue Bereiche der Informatik entdecken und erlernen, wie zum Beispiel die Nutzung von Docker oder Machine Learning mit TensorFlow.
Generell können wir abschließend sagen, dass wir durch das Projekt in den verschiedensten Bereichen einiges gelernt haben und froh sind, an diesem Projekt teilgenommen zu haben.

Literaturverzeichnis

Arthur Juliani (2016): Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks. medium.com. Online verfügbar unter https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0, zuletzt aktualisiert am 25.08.2016, zuletzt geprüft am 17.01.2021.

Edsger W. Dijkstra (1972): The Humble Programmer, 1972. Online verfügbar unter https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html.

jetbrains.com (2021): PyCharm Scientific Mode. Hg. v. JetBrains. https://www.jetbrains.com. Online verfügbar unter https://www.jetbrains.com/help/pycharm/matplotlib-support.html, zuletzt aktualisiert am 2021, zuletzt geprüft am 17.01.2021.

K. R. Srinath (2017): Python - The Fastest Growing Programming Language. In: *International Research Journal of Engineering and Technology*, S. 354–357. Online verfügbar unter https://d1wqtxts1xzle7.cloudfront.net/55458585/IRJET-V4I1266.pdf?1515226715=&response-content-disposition=inline%3B+filename%3DPython_The_Fastest_Growing_Programming_L.pdf&Expires=1610810535&Signature=Mi1q-sk6ayhXxW-IGxKcNQQ9DJN8Ydme-8KWYWU5GnYoQQ21odJXSNg-zevCeLu0Fb85pV3vEgMS5fBWCBqnKKWgE-OFxmF3k8XJ5eypq8nLuPcm0A9Fg1w2aWvB368FlwZLDP48q1wzonoHLC14et5CzgFt-Tzv4hQVmW7jCEMDMXCXmOYpgXL0k6OiNEuZrKSgLz4OhN0Xv8V2mEIn-Spwp6xvzsjB1GzzEMN2Kx2Fz7V9J3i35ckaOQ2ifq4WXs5C~OKXkqp08mnb02QBaid2kThPomvUFEDOGMxKmLiPyDtp0WcU6i4-Z3641lTY29ahy~BBfybop90aMZKGyP-drEw__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA, zuletzt geprüft am 17.01.2021.

Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar (2018): Foundations of Machine Learning. A new edition of a graduate-level machine learning textbook that focuses on the analysis and theory of algorithms.: MIT Press.

Paul E. Black (2019): Dictionary of Algorithms and Data Structures. Manhattan distance. https://xlinux.nist.gov/dads/. Online verfügbar unter https://xlinux.nist.gov/dads/HTML/manhattanDistance.html, zuletzt aktualisiert am 11.02.2019.

Pierre Carbonnelle (2020): PopularitY of Programming Language. The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched on Google. Hg. v. Pierre Carbonnelle. Online verfügbar unter https://pypl.github.io/PYPL.html, zuletzt aktualisiert am 01.2021, zuletzt geprüft am 17.01.2021.

Prof. Dr. Marco Lübbecke (2018): Branch-and-Bound-Verfahren. Definition: Was ist "Branch-and-Bound-Verfahren"? Hg. v. Springer Gabler. Online verfügbar unter https://wirtschaftslexikon.gabler.de/definition/branch-and-bound-verfahren-28551/version-252179, zuletzt geprüft am 17.01.2021.

Randall Munroe (2021): about xkcd. Hg. v. Randall Munroe. https://xkcd.com. Online verfügbar unter https://xkcd.com/about/, zuletzt aktualisiert am 2021, zuletzt geprüft am 17.01.2021.

Anhang

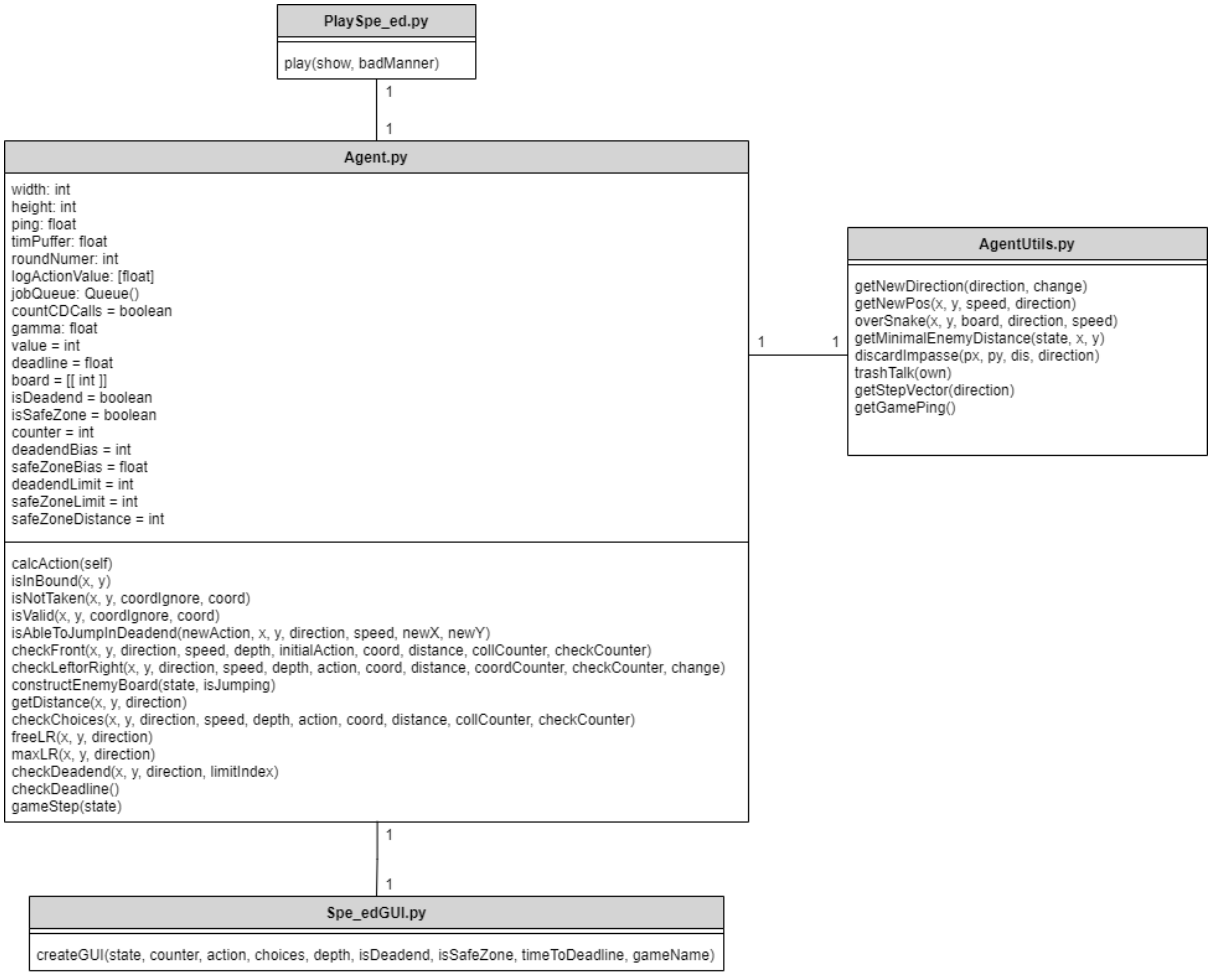


Abbildung 10: Klassendiagramm DommeAI

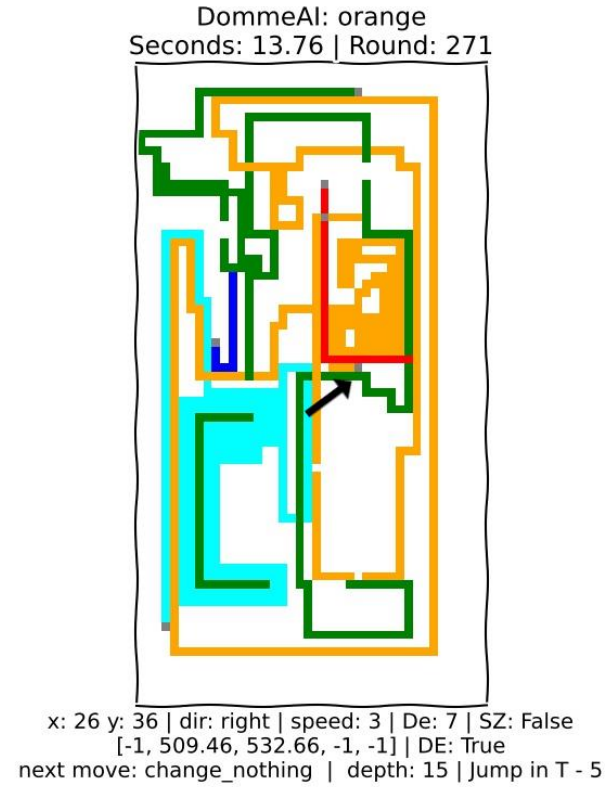


Abbildung 11: Schlange in einer Sackgasse vor einem Sprung

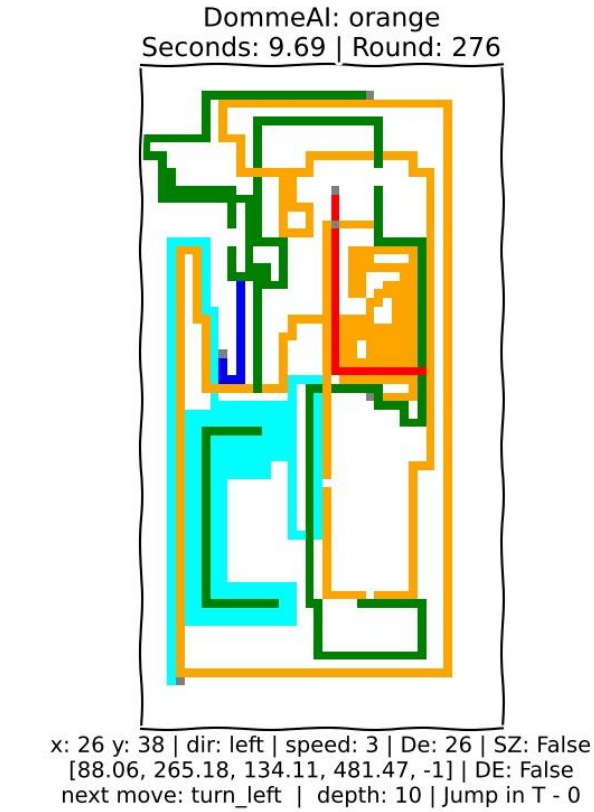


Abbildung 12: Schlange nach dem Sprung aus einer Sackgasse

Abbildung 13: Schlange in einer safeZone

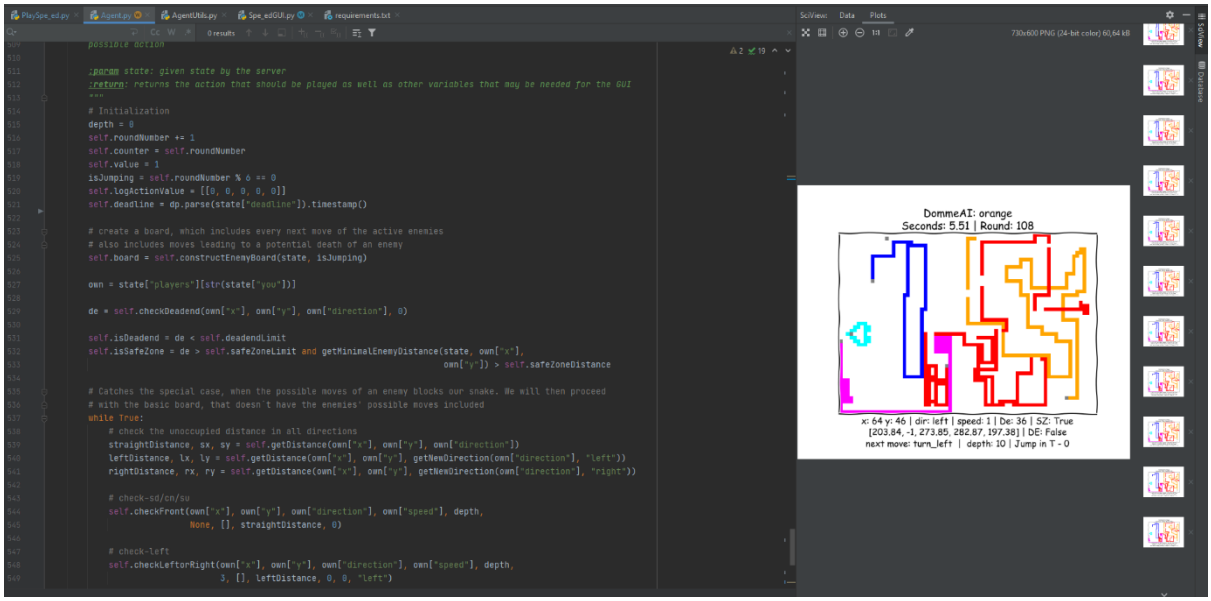


Abbildung 14: Ansicht im SciView-Feature