Dokumentation der Gruppe SnakeInc beim InformatiCup2021

Sebastian Diers, Jannes ter Veen, Jordi Welp, Joost van Mark

17 Januar 2021

Contents

1 Einleitung

Im Wettbewerb Spe_ed des InformatiCup2021 [IFC] bestand die Aufgabe in der Entwicklung eines Teilnehmers an dem Computerspiel Spe_ed. In diesem Dokument wird zunächst auf die Planung und unsere Vorüberlegungen eingegangen. Anschließend werden mögliche Lösungsansätze beschrieben. Dafür werden zunächst die verworfenen Ansätze aufgezeigt und anschließend der umgesetzte Lösungsansatz ausführlich beschrieben. Dabei wird zunächst die theoretische Idee und anschließend die Umsetzung vorgestellt, sowie ein Ausblick auf eine mögliche Fortsetzung des Projekts aufgezeigt. Zuletzt wird auf die entwickelten und verwendeten Hilfstools eingegangen und das Vorgehen im Rahmen der Codequalität beschrieben.
Unser entwickelter Teilnehmer wird im folgenden als "unser Spieler" bezeichnet.

2 Planungen und Überlegungen

Zu Beginn des Projekts haben wir ein Projektmanagement und eine Arbeitsweise festgelegt. Da die Thematik einen Bot oder eine künstliche Intelligenz zu entwickeln für uns neu war, haben wir uns für eine agile Entwicklungsmethode entschieden und uns auf das uns bekannte Scrum geeinigt. In diesem Sinne wurden zwei wöchentliche online Meetings mit verbindlichen Weekly-Scrums und Tagesordnungen vereinbart. Zur Organisation der Meetings und dessen Tagespunkte, sowie zur Verwaltung und Entwicklung der Software, wurde Gitlab genutzt. Unsere ersten Meetings dienten ausschließlich zur Findung von verschiedenen Lösungsansätzen und dem Sammeln von Ideen und Konzepten. Anschließend wurden die Ansätze recherchiert und auf eine mögliche Umsetzung geprüft. Gütekriterien waren bei der Analyse die mögliche Umsetzbarkeit in Abhängigkeit unserer Ressourcen und wissenschaftliche Erkenntnisse über die Erfolge der Ansätze im Zusammenhang mit Mehrspieler-Problemen.

3 Verworfene Ansätze

3.1 Neuronales Netz

Neuronale Netze bieten vielfältige Möglichkeiten eine künstliche Intelligenz zu entwickeln. Da wir die Komplexität durch den Mehrspieler-Charakter und die Möglichkeit des Springens als hoch betrachten, vermuten wir, dass eine Lösung über ein angelerntes neuronales Netz nicht die beste Lösung ist. Rückschlüsse darauf konnten wir in [MSA16] ziehen, die einen Q-Learning- Ansatz für ein Mehrspieler-Snake-Spiel entwickelt und evaluiert haben. Diese beschreiben den Erfolg in großer Abhängigkeit

den gelernten Situationen. Da für uns keine gleichbleibende Situationen vorhanden sind und sich die Gegner wechseln, haben wir den Ansatz ausgeschlossen.

3.2 MaxiMin

Mit dieser Idee sollte ein Suchbaum aus allen möglichen Spielzügen der Spieler aufgespannt werden. In dem Suchbaum sollte dann bewertet werden, welchen Zug die jeweiligen Spieler mit welcher Wahrscheinlichkeit wählen. Für unseren Spieler sollten die Spielzüge dann anhand des Nutzen eines Spielzuges bewertet und mit der Wahrscheinlichkeit des Eintretens der Spielzüge der Gegner verrechnet werden. Daran sollte der Algorithmus den Spielzug auswählen, der mit der größten Wahrscheinlichkeit den größten Nutzen hat.
Nach einigem Experimentieren mit der Berechnung des Suchbaumes, wurde dieser Ansatz nicht weiterverfolgt, da die Komplexität der Berechnung zu hoch ist. Sie liegt im schlechtesten Fall bei $\mathcal{O}((z^s)^t)$, mit z Anzahl der möglichen Züge eines Spieler, hier 5, s die Anzahl der Spieler, hier 6, und der Tiefe der Suche t. Damit ist die Anzahl der Äste in dem Suchbaum auch für kleine Suchtiefen sehr groß und im Prototyp nicht mit der benötigten Geschwindigkeit berechenbar.

3.3 Mustererkennung

Eine weitere Idee lag in der Entwicklung einer Mustererkennung, um in den Zügen der anderen Spieler Muster zu finden. Dazu sollten alle Spiele auf dem offiziellen Server aufgezeichnet werden, an denen dann eine künstliche Intelligenz trainiert werden sollte. Auf diese Weise sollten Muster in den Spielzügen anderer Spieler erkannt werden und Züge vorhergesehen werden.
Schwierigkeiten bei diesem Ansatz bestehen, darin die aufgezeichneten Spieldaten und auch die Daten des aktuellen Spiels in ein über alle Spiele vergleichbares Format zu überführen. Die Spielfelder selbst sind aufgrund ihrer variierenden Größe und Spieleranzahl schwer vergleichbar.
Die KI sollte unter Berücksichtigung der Spielsituation und der vorhergehenden Züge eine Wahrscheinlichkeit dafür Abgeben, welchen Zug ein Gegner macht, und damit unsere Steuerung in Konfliktsituationen unterstützen. Weitere Schwierigkeiten sind auch, dass die Mustererkennung ihren Nutzen vorraussichtlich erst dann entfalten kann, wenn es zu Konflikten mit andere Spielern kommt. Das setzt aber Voraus das andere Systeme vorhanden sind, die die Steuerung der Schlange in Situationen ohne Konflikt übernehmen und ihr überleben in statischen Situationen garantieren.

4 Unser Ansatz

Die grundlegende Idee unseres Systems liegt darin, jeder Zelle des Spielfeldes einen Kostenwert zuzuweisen, der das Risiko und den Nutzen des Betretens und Durchqueren des Feldes darstellt. Die Schlange soll dann den Pfad mit den geringsten Kosten beziehungsweise Risiko nehmen.
Das System erlaubt unterschiedliche Analysen modular und unabhängig von einander auf dem Spielfeld durchzuführen. Informationen in Form von Risiken können dabei über mehrere Spielzüge gespeichert werden. Auf diese Weise sollen möglichst viele Informationen für die Auswahl der Züge einbezogen werden.

5 Unser Ansatz - Das Basissystem

Die Grundstruktur unseres Ansatzes besteht aus dem EvaluationBoard, dem BoardAnalyzer und den Zellen (`Cell[][]`) sowie den Analysemodulen. Im EvaluationBoard werden die Updates der einzelnen Runden verarbeitet (`.update()`) und damit die Zellen aktualisiert. Der BoardAnalyzer bildet die Schnittstelle zu den Analysemodulen. Von hier werden in jeder Runde die gewünschten Algorithmen zur Analyse des Spielgeschehens innerhalb der `analyze(...)`- Methode ausgeführt und nach abschließen der Runde mit der `prepareNextPhase()`-Methode auf die nächste Runde vorbereitet. Die Zellen bilden den Speicherort für die Ergebnisse der Analysen. Jede Zelle kann je nach analysierten Umgebungsbedingungen angepasst werden. Diese Anpassungen werden in den Attributen der Zellen gespeichert. Über die Methode `getRisks()` erhält man das Risiko einer Zelle, in dem alle gespeicherten Risiken miteinander multipliziert werden. In der MoveCalculation werden dann die einzelnen Werte der Zellen genutzt, indem die Risiken derer, die man in einer Folge von Aktionen betreten würde, miteinander multipliziert. Auf diese Weise erhält man für jede Kombination an Zügen einen Wert in Abhängigkeit der durchlaufenden Zellen. Wichtig ist im Zusammenhang dieser Struktur, dass der neutrale Wert einer Zelle bei 1 liegt. Ein Erhöhen des Wertes auf über 1 sorgt dementsprechend für eine Erhöhung des Risikos, während das Senken eines Wertes auf unter 1 das Durchlaufen einer Zelle attraktiv macht. Durch diese Bewertung lassen sich verschiedene Kombinationen von zukünftigen Spielzügen gegeneinander abwägen.

Das Klassendiagramm zeigt die relevantesten Strukturen, Methoden und Attribute.

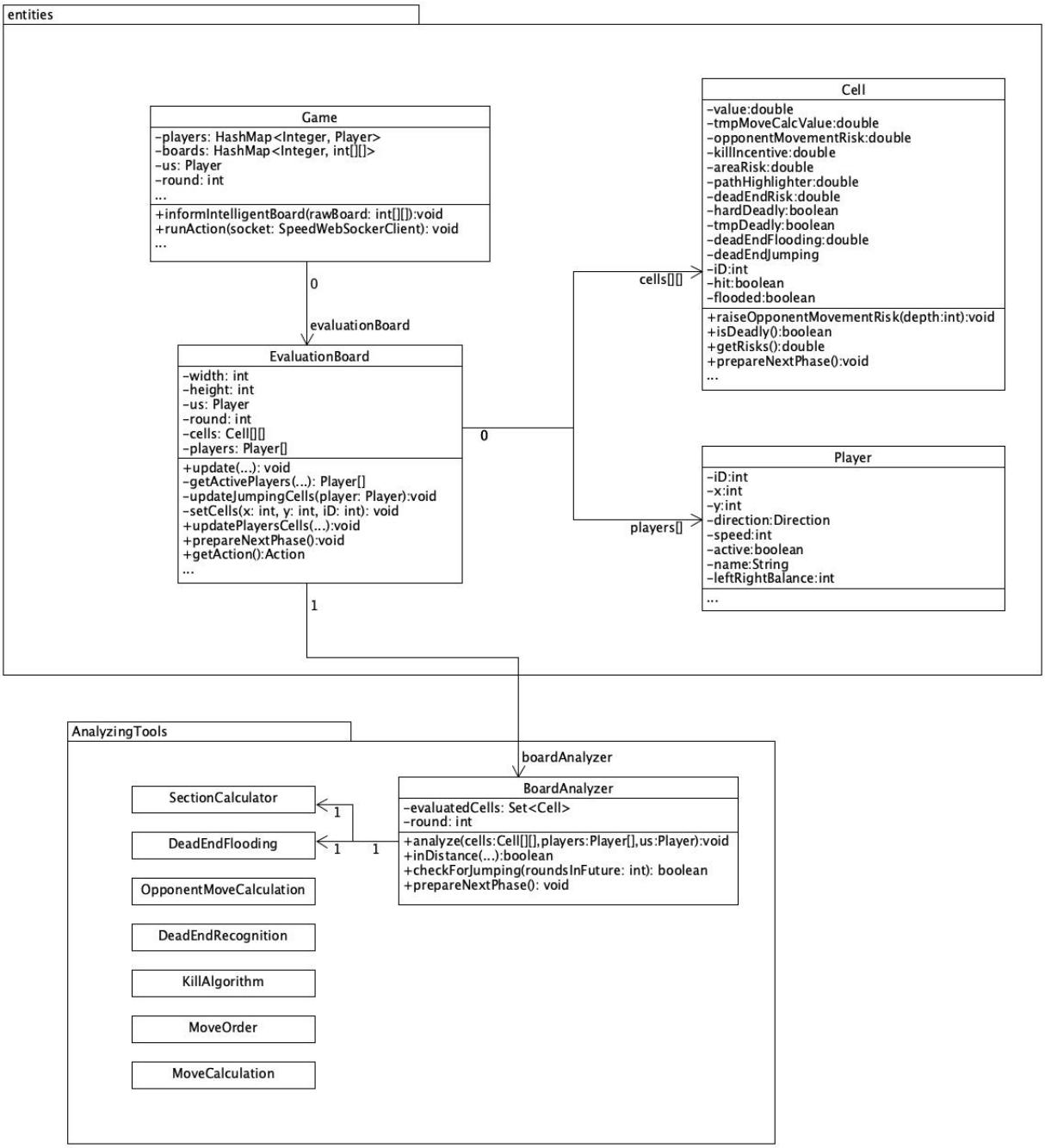

Figure 1: Klassendiagramm des Basissystems

Dieser Ansatz verspricht uns eine deutlich höhere zeitliche Performance als ein MaxiMin-Ansatz, da viele Interaktionen nicht in der Tiefe des Suchbaums berechnet werden müssen. Dies führt dazu, dass wir die eingesparte Performance dazu nutzen können, vielfältigere Informationen aus dem Spielfeld zu sammeln und zu

analysieren und letztendlich in unserer Berechnung zu berücksichtigen. Sollten trotzdem Performance-Probleme auftreten, bietet der Ansatz zudem die Möglichkeit besonders aufwendige Analysen erst nach der Berechnungen des Zuges auszuführen. Auf diese Weise lässt sich ein zu spätes Antworten an den Server verhindern.
Schwächen liegen vor allem in der nicht optimal beachteten Interaktion zwischen den gewonnen Informationen. Diese Problematik kann durch ein möglichst gutes Balancing der Bewertungen der einzelnen Informationen ausgeglichen werden. Eine weitere Schwäche könnte in der direkten Interaktion mit anderen Teilnehmern liegen, die sich auf Konfrontation spezialisiert haben.

5.1 Module

Die Informationen aus dem Spielgeschehen werden durch getrennte Algorithmen analysiert und in den Zellen gespeichert. Jeder Algorithmus wird durch ein Modul beschrieben.

5.1.1 MoveCalculation

Die Grundidee der MoveCalculation liegt in der Nutzung der Bewertungen der einzelnen Zellen des Spielfeldes. Dies geschieht, indem alle eigenen Spielzüge bis in eine vorgegebene Tiefe vorausberechnet werden. Dabei wird für jeden Pfad (Kombinationen von aufeinander folgenden Spielzügen bis zur gewählten Tiefe) ein Wert errechnet. Dieser ergibt sich durch die Multiplikation der Risiken der möglicherweise von unserem Spieler belegten Zellen. Diese Berechnung wird für jeden Pfad der gewählten Tiefe durchgeführt und verglichen. Der Pfad mit dem niedrigsten Risiko wird am Ende ausgewählt und führt zur Auswahl der Aktion für die aktuelle Spielrunde. Diese Berechnung findet jede Spielrunde erneut statt.
Der Algorithmus wurde mittels einer Rekursion implementiert, sodass für jede mögliche Aktion im Spiel, jeweils alle folgenden möglichen Aktionen bis zur Endtiefe berechnet werden. Das Nutzen einer Rekursion ermöglicht es zudem bei tödlichen Zügen, die Berechnung frühzeitig abzubrechen und somit die Rechenleistung zu verringern.

5.1.2 OpponentMoveCalculation

Der OpponentMoveCalculation- Algorithmus berechnet die potenziellen Risiken durch andere Spieler. Dafür werden alle ihre möglichen zukünftigen Züge berechnet und je nach Tiefe als Risiko in die zugrundeliegenden Zellen eingepreist. Zellen, die nicht erreichbar sind werden dabei nicht bewertet.

Der Algorithmus könnte durch eine flexiblere Bewertung der einzelnen Züge der Gegenspieler verstärkt werden. Anstatt lediglich die Tiefe der Rekursion als Bewertungsmaßstab heranzuziehen, könnte man zusätzlich die Attraktivität eines Zuges für den Gegenspieler einbeziehen. Kann ein Gegenspieler beispielsweise nur zwischen wenigen Aktionen wählen, weil die anderen versperrt sind, so werden diese Aktionen wahrscheinlicher gewählt, als wenn keine der anderen Aktionen versperrt ist.

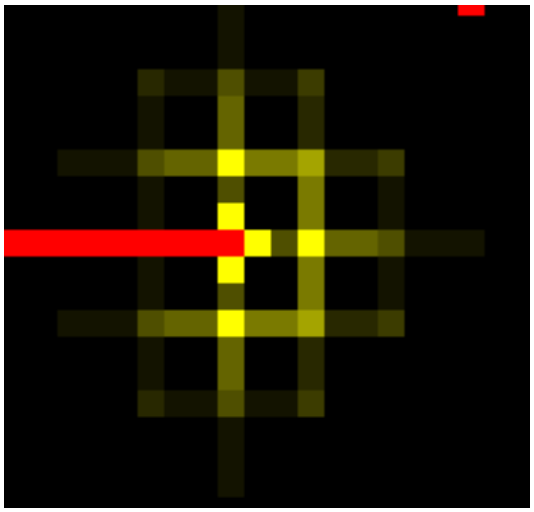

Figure 2: Berechnung des Risikos durch gegnerische Spielzüge.

In der Abbildung sind die Verteilungen der berechneten Risiken zu erkennen. Je gelber das Feld desto höher ist die Risikobewertung. Die roten Felder sind schon von der Schlange belegt.

5.1.3 Pathfinder

Für unsere Pfad-Hervorhebung und unseren Kill-Algorithmus müssen zeitweilig die schnellsten Wege zu bestimmten Zellen auf dem Spielfeld gefunden werden. Hierzu wurde die abstrakte Klasse Pathfinder entwickelt. Hier nutzen wir eine A*-Suche zur Wegfindung. Dieser basiert auf einer Implementation von RosettaCode [RTC] und wurde für unsere Zwecke angepasst.
Der Pathfinder selbst ist aufgrund seiner Struktur erweiterbar und könnte grundsätzlich mit Algorithmen wie Dijkstra, BFS oder DFS erweitert werden. Der A*-Algorithmus bietet sich aber besonders aufgrund seiner Geschwindigkeit an. Vor allem in größeren Grid-Strukturen hat dies ein Vorteil [CPA19].

5.1.4 StartingStrategy

Mit der StartingStrategy war eine Strategie geplant, die zu Beginn des Spiels ausgeführt wird und zu einer möglichst guten Ausgangssituation führen sollte. Im Verlaufe der Entwicklung stellte sich die Frage, ob eine solche Strategie einen Nutzen hat, wenn die Spieler die Möglichkeit haben zu Springen. Dies entkräftet beispielsweise das Abtrennen eines Bereiches der Spielfläche. Durch die gewählte Herangehensweise möglichst viele Informationen aus dem Spielfeld und dem Spielgeschehen miteinzubeziehen, kommt hinzu, dass unser Spieler bereits frühzeitig zielgerichtet agieren kann.

5.1.5 Area-Evaluation

Die Area-Evaluation soll unserem Spieler leere Bereiche auf dem Spielfeld schmackhaft machen um mögliche Engpässe frühzeitig zu erkennen.
Hierzu wird das Spielfeld in einzelne Sektionen aufgeteilt. In den Sektionen berechnen wir in mehreren Iterationen die Prozentzahl freier Felder in der Sektion. Hier fangen wir mit vielen kleinen Sektionen an und vervierfachen die Größe der Sektion mit jeder Iteration. Jeder Wert einer neuen Iteration wird dabei mit $\frac{3}{4}$ der vorherigen bewertet, sodass die kleinste Sektion die größte Gewichtung hat. Dieses Prinzip ermöglicht es uns kleinere Sektionen mit viel Freiraum innerhalb einer großen Sektion mit wenig Freiraum gegenüber Sektionen mit viel Freiraum innerhalb einer großen Sektion mit viel Freiraum schlechter zu bewerten. Durch das Iterative Prinzip verhindern wir, dass falsche Schlüsse gezogen werden. Würden wir nur eine oder zwei Iteration nutzen, so würden die Sektionen nur ein kleines Feld abdecken, was die Aussagekraft verringern würde.

Figure 3: Berechnung der Sektions-Risiken.

Der Algorithmus im Hintergrund läuft bei der SectionCalculation. Hier wird das Feld zuerst auf die größte Auflösung gerastert und innerhalb einer Schleife, die freien und belegten Felder innerhalb der einzelnen Sektionen gezählt.
In einem nächsten Durchlauf werden hieraus die Prozentwerte für die Sektionen berechnet. In einem dritten Durchlauf werden über `createDeepiterations()` zuerst die Prozentangaben der jeweils kleineren Auflösungen iterativ berechnet. Anschließend werden die Werte in einem neuen Durchlauf den jeweiligen Sektionen hinzugefügt. Hierzu werden die Werte zuerst multipliziert und zum Schluss durch den maximalen Prozentwert geteilt. Damit ergibt sich das finale Sektions-Grid mit den jeweiligen Werten für die einzelnen Sektionen. Die Auflösung entspricht dabei der größten Auflösung. In einem weiteren Durchlauf werden außerdem die minimalen und die maximalen Prozentwerte errechnet. Anschließend wird in der Methode `rankAreaRiskCells()` für alle Zellen ein Wert gesetzt. Wie bei der Pfad-Hervorhebung wird hierbei der minimale Prozentwert von dem Prozentwert der Sektion subtrahiert und durch den Abstand zwischen dem minimalen und maximalen Prozentwert der Karte dividiert. Über eine lineare Interpolation wird dann

das Risiko der Zelle anhand der Skala aus der vorherigen Rechnung berechnet und gesetzt.

5.1.6 Pfad-Hervorhebung

Die Pfad-Hervorhebung zeigt unserem Spieler den Weg in bessere Sektionen, in denen prozentual mehr Felder frei sind.
Hierzu wird mithilfe der Area-Evaluation das Feld mit der besten Bewertung (den meisten freien Feldern) genommen und versucht ein Pfad von der aktuellen Position unseres Spielers zu diesem Feld zu finden. Sollte ein Weg möglich sein, so erhalten die ersten 12 Felder dieses Weges, beginnend vom Spieler, einen Anreiz, also einen Kostenwert der kleiner als 1 ist. Je nach Unterschied zwischen dem Wert des aktuellen Feldes und dem Feld zu dem der Pfad lenkt, wird der Anreiz der Pfad-Felder stärker oder schwächer, sodass es bei geringen Unterschieden zwischen den Feldern nicht zu sprunghaftem Wechseln zwischen diesen kommt. Sollte kein Weg gefunden werden, wird keine Aktion getroffen und auch kein Wert gesetzt.

Der Algorithmus läuft innerhalb des SectionCalculators und basiert auf den Werten aus der Area-Evaluation. Bei der Berechnung der Prozentangaben für die Area-Evaluation werden die Werte bestX und bestY errechnet, welche die Koordinaten der besten Sektion auf der Karte darstellen. In der Methode rankPathHighlightCells() wird eine freie Position innerhalb der gefunden Sektion gesucht. Sollte kein freies Feld gefunden werden, so wird an dieser Stelle abgebrochen. Ansonsten wird mithilfe des A*-Suchalgorithmus versucht ein Weg zwischen der aktuellen Position unseres Spielers und der gefundenen freien Zelle zu finden. Sollte ein Weg gefunden worden sein, so wird der Vorteil für den Weg berechnet. Hierzu wird der minimale Prozentwert von dem Prozentwert unserer Sektion angezogen und der erhaltene Wert durch den Differenz zwischen dem maximalen Prozentwert und dem minimalen Prozentwert geteilt. Über lineare Interpolation wird mithilfe dieser erarbeiteten Skala dann der jeweilige Anreizwerte für den Pfad ermittelt. Im Anschluss werden für die ersten 12 Felder beginnend von der Position unseres Spielers die Anreizwerte gesetzt. Vor jedem Durchgang werden die Werte wieder auf den Standardwert zurückgesetzt.

5.1.7 Berechnen von Angriffspunkten

Die Berechnung von Angriffspunkten erfolgt über den KillAlgorithm. Angriffspunkte werden als Zellen verstanden, die statt eine Erhöhung des Risikos, eine Senkung beziehungsweise ein Incentive erhalten. Dies geschieht durch die Setzung des At-

tributes killIncentive = true, welches die Gesamtbewertung der Zelle verringert und somit einen Anreiz in der MoveCalculation für das Ablaufen der Zelle bietet. Die MoveCalculation sucht den Pfad, der über die Zellen mit der geringsten Gesamtbewertung verläuft, weshalb eine Zelle mit einem Wert kleiner als 1 zu einer Bevorzugung führt.
Der Algorithmus funktioniert nach dem Prinzip dem Gegenspieler den Weg abzuschneiden, falls dieser dadurch in einer Sackgasse gefangen wäre. Ist ein Gegenspieler in Reichweite, nutzt der Algorithmus den A*-Algorithmus um einen Pfad zum Kopf des Gegenspielers zu finden. Dieser wird genutzt, um den Raum um den Gegenspieler in zwei Bereiche, zwei potenzielle Sackgassen, aufzuteilen. Durch Flooding wird anschließend auf Sackgassen und deren Größen geprüft, um somit die vielversprechendste Angriffsrichtung zu ermitteln.

Da die Angriffspunkte jeden Zug neu berechnet werden und stets in einer Entfernung von der doppelten Geschwindigkeit des Gegners liegen, wird versucht das eigene Risiko in den Gegner zu steuern möglichst gering zu halten. Gewünscht ein langsames Einkesseln, beziehungsweise dem Gegner den Weg abzuschneiden und abzubrechen, falls es für unseren Spieler tödlich enden würde.
Schwierig gestaltet sich, einerseits die Sackgassen präzise zu erkennen, und andererseits die Angriffsrichtung in Abhängigkeit von der eigenen Position zu bestimmen. Die Erkennung der Sackgassen wird vor allem durch das Finden eines geeigneten Punktes zum Starten des Floodings erschwert, da eine Vielzahl von Konstellationen des Kopfes des Gegenspielers zur Umgebung möglich ist. Um möglichst wenig fälschliche Sackgassen zu identifizieren wird deshalb lediglich das Flooding seitlich des Kopfes ausgelöst. Die zweite Problematik liegt in der Vielzahl der möglichen Konstellationen zwischen den Positionen von unserem und dem gegnerischen Spieler, sodass es dazu kommen kann, dass die Angriffspunkte für unseren Spieler nur schwer zu erreichen sind. Relativiert wird dieses Problem, durch die beschränkte Sichtweite unseres Spielers. Liegen die Angriffspunkte außerhalb dieser, so bleiben Sie unbeachtet. Diese Lösungen für die Problematiken scheinen im Test zu funktionieren, bieten allerdings auch ein Verbesserungspotenzial.
Im Gegensatz zu dieser Lösung wäre auch ein Ansatz möglich gewesen, Konstellationen zu erkennen, in denen man über ein bestimmtes Handeln den Gegner mit einer sicheren Wahrscheinlichkeit eingrenzen könnte. Dieser Ansatz wurde allerdings nicht verfolgt.

5.1.8 Berechnen von Sackgassen

Um Sackgassen zu erkennen wird jeder Zug, wenn er möglich ist, auf einer Kopie der Spielfeldes getestet. Für diesen Test werden die Zellen als tödlich markiert und alle Zellen um die entstandenen Zellen berechnet. Für diese Zellen wird einmal für jeden Bereich, wobei eine tödliche Zelle einen neuen Bereich definiert, eine Methode ausgeführt, die alle Bereiche um eine Zelle erkennt. Für die gefundenen, nicht tödlichen Zellen, wird der Flooding-Algorithmus ausgeführt, der die Anzahl an Zellen in der Sackgasse berechnet. Anhand dieser Anzahl werden alle Zellen in der Sackgasse bewertet. Der Wert hängt logarithmisch von der Anzahl der Zellen ab.

5.1.9 Berechnen von Sackgassen 2

Die Sackgassen-Erkennung 2 soll zusätzliche problematische Sackgassen erkennen und dient als Grundlage für die Sackgassen-Befreiung.
Der Algorithmus basiert grundsätzlich auf dem Prinzip des Floodings. Hierbei werden zunächst die möglichen fünf Züge abgewägt. Für alle Möglichen Züge wird beginnend von der Zug-Ausgangsposition der Bereich geflutet. Hierzu werden zunächst der Weg zur Zug-Position von der aktuellen Position sowie die Position selbst als Hit markiert und anschließend geflutet. Sollten dabei weniger Blöcke als definiert gefunden werden, markiert der Algorithmus diese Blöcke als Sackgasse, indem alle Felder markiert werden. Um zu verhindern, dass ein anderer möglicher Zug nicht schlechtere Felder bekommt, werden die zurückliegenden Felder, welche nicht durch den Sackgassen-Zug direkt erreicht werden niedriger bewertet als die direkte Zug-Position. So kann unserer Spieler über andere Züge, mit denen er die “Sackgasse” verlassen könnte, dennoch in die “Sackgasse” gelangen. Ein Beispiel hierfür wäre ein Change-Nothing, welches eine zwei Block-Breite Nische verdeckt, während ein Turn-Left die eine Seite nur blockiert, sodass er auf dem nebenliegenden Block wieder entkommen könnte. Die Bewertung der Sackgassen ist zudem abhängig der freien Blöcke. Sollte unser Spieler also zwischen zwei Sackgassen entscheiden müssen, würde er die Sackgasse mit der größeren Anzahl verfügbarer Blöcke wählen.
Zuerst werden hierbei die Positionen der nächsten Züge auf Möglichkeit geprüft. Für alle möglichen Züge wird anschließend ein Flooding-Algorithmus benutzt, der das eigene Feld bereits als Hit wertet und von dieser Zelle aus die nächsten Nachbarn nimmt. In jeder Iteration wird von allen Blöcken die Nachbarn genommen, die frei sind und nicht durch ein vorheriges Auftreffen als Hit markiert sind. Dies wird solange durchgeführt, bis es entweder keine neuen Nachbarn gibt oder die Zahl der Blöcke die definierte Anzahl an Blöcken erreicht hat.
Im Falle, dass die Anzahl der Blöcke geringer als die definierte Anzahl ist, werden

alle Blöcke innerhalb der Sackgasse mit einem Wert abhängig der freien Blöcke bewertet. Dieser entspricht der Anzahl der Blöcke geteilt durch die maximalen Blöcke, die eine Sackgasse haben darf. Dieser Wert wird als Skala für eine lineare Interpolation verwendet, um einen Wert zwischen 1.0 und dem Sackgassen-Wert zu berechnen. Der Block der nächstmöglichen Zug-Position wird dabei um einen definierten Wert schlechter bewertet.

5.1.10 Sackgassen- Befreiung

Die Sackgassen-Befreiung erkennt, wenn unser Spieler in eine Sackgasse gefahren ist und versucht diesen durch gezielte Bewertung von Zellen, mithilfe des Springens, aus dieser Sackgasse zu lenken.
Hierzu wird mithilfe eines Flooding-Algorithmus der Sackgassen-Erkennung 2 der umliegende Bereich geflutet. Sollte der maximal ausfüllbare Bereich eine definierte Anzahl an Zellen unterschreiten, gilt der Bereich als Sackgasse. Grundsätzlich würde hier die Sackgassen-Erkennung 2 bereits eingreifen um eine Sackgassen-Situation zu verhindern. Sollte dies aber nicht möglich sein, bzw. diese zwischen zwei Sackgassen abwägen, muss eine andere Möglichkeit gefunden werden. In diesem Fall werden alle Zellen außerhalb der eigentlichen Sackgasse, die nicht belegt sind, mit einem sehr hohen Incentive-Wert belegt. Dies macht das Erhöhen der Geschwindigkeit verbunden mit einem Sprung attraktiv. Sollte nun eine Möglichkeit bestehen in ein anderes Feld zu springen, so wird der Weg aufgrund des hohen Incentives besser bewertet als die anderen und der Spieler springt aus der Sackgasse.
Dies wird durch einen Flooding-Algorithmus realisiert, der das eigene Feld bereits als Hit wertet und von dieser Zelle aus die nächsten Nachbarn nimmt. In jeder Iteration werden von allen Blöcken die Nachbarn genommen, die frei sind und nicht als Hit markiert sind. Dies wird solange durchgeführt, bis es entweder keine neuen Nachbarn gibt oder die Zahl der Blöcke die definierte Anzahl an Blöcken erreicht hat.
Im Falle, dass die Anzahl der Blöcke geringer als die definierte Anzahl ist, werden alle Blöcke innerhalb der Sackgasse mit 1.0 bewertet und alle außerhalb der Sackgasse erhalten einen sehr hohen Incentive-Wert.

Eine geplante Erweiterung, die zeitlich nicht mehr umgesetzt werden konnte, sollte es ermöglichen die einzelnen freien Bereiche anhand der Größe zu bewerten, sodass nicht in eine kleinere Sackgasse gesprungen wird, die zum Tod führen würde. Hier war geplant die freien Blöcke iterativ zu durchgehen und bei jedem Block ein Flooding durchzuführen. Die gesammelten Blöcke würden dann einer Liste hinzugefügt

und gespeichert werden. Wenn alle Blöcke geflutet worden wären und als erkannt gelten, würde eine Skala anhand der größten und der kleinsten Bereiche erstellt werden. Mithilfe dieser würde eine lineare Interpolation hinsichtlich des Incentive-Werts erstellt werden, welche Zellen mit weniger Platz schlechter bewerten würde. Somit könnten die Sackgassen abgewogen werden.

5.1.11 MoveOrder

Die Klasse MoveOrder ist dazu da, bei gleich bewerteten Zügen festzulegen welcher gefahren wird. Dabei kann eingestellt werden werden, ob der Algorithmus versuchen soll Spiralen zu fahren und auf welches Geschwindigkeitsintervall er sich einpendeln soll. Die MoveOrder fällt weniger ins Gewicht, seitdem mehr Analysen das Spielfeld flächendeckend bewerten. Sie ist so eingestellt, dass die Schlange Spiralen vermeidet, und eine Geschwindigkeit von 1 bis 2 fährt.

6 Unser Ansatz - Erfahrungen

Mit zunehmender Menge an Analysen und deren Komplexität wird das Ausbalancieren der Bewertungen gegeneinander schwierig, sodass eine relevantere Analyse eine weniger relevante Analyse überdecken und zu einer Fehlsteuerung führen kann. Anders als erwartet ist der direkte Konflikt mit anderen Schlangen eher Zweitrangig, viel wichtiger und auch schwieriger als erwartet ist das Überleben in Situationen ohne Konflikt. Vor allem, dass die Schlange nicht in eine Sackgasse fährt oder sich den eigenen Weg abschneidet. Wir hatten erwartet dieses Aufgabe mit einer tiefen MoveCalculation zu lösen. Stattdessen war allerdings eine Sackgassenanalyse notwendig.

7 Ausblick

In manchen Modulen wurden bereits Verbesserungsideen oder Alternativlösungen aufgezeigt. Diese können perspektivisch umgesetzt werden. Des Weiteren wäre eine weitere Ausbalancierung der Risikowerte auf Grundlagen von Statistiken über die Ausgänge der Spiele ein nächster Schritt. Die Offenheit des Ansatzes zur Implementierung neuer Analysetools ermöglicht es zudem weitere vielfältige Informationen aus dem Spielgeschehen zu analysieren.

8 Tools

8.1 CSV-Generator

Der CSV-Generator ist ein zusätzliches Tool, welches Auswertungen hinsichtlich der Performance über mehrere Spiele hinweg erstellen soll. Hierzu können Export-Files in den Unterordner exports gelegt werden und der CSV-Generator generiert aus allen json-Exports eine data.csv.
Diese enthält folgende Werte:
Name: Name des Spielers (Dieser ändert sich zwar regelmäßig, bleibt aber für einen kurzen Zeitraum gleich, sodass Teams beobachtet werden können [IFC])
Games: Anzahl der gespielten Spiele über alle Exports hinweg
First: Anzahl der Spiele, die mit der Position 1 abgeschlossen wurden
Second: Anzahl der Spiele, die mit der Position 2 abgeschlossen wurden
Third: Anzahl der Spiele, die mit der Position 3 abgeschlossen wurden
Fourth: Anzahl der Spiele, die mit der Position 4 abgeschlossen wurden
Fifth: Anzahl der Spiele, die mit der Position 5 abgeschlossen wurden
Sixth: Anzahl der Spiele, die mit der Position 6 abgeschlossen wurden
TrustedValue: Gibt an, wie zuverlässig die Positions-Angaben eines Spielers sind. Spieler mit einem hohen Wert haben tendenziell auch bei Spielen mit vielen Gegnern eine gute Position erreicht. Der Wert wird wie folgt berechnet: Für jedes gespielte Game wird der Wert aus 1 - (Position / Anzahl Spieler) hinzugefügt. Am Ende wir dieser Wert durch die Anzahl der Spiele geteilt.
MedianPlayers: Durchschnittliche Anzahl von Spielern über alle gespielten Runden des Spielers
TrustedScoreEnemies: Ein grober Wert, wie der eigene Bot gegenüber anderen starken/schwachen Bots abgeschnitten hat. Ist der Wert hoch, so schlägt der Bot auch andere starke Kontrahenten.
Anhand dieser Werte ist ein grober Vergleich der Leistung gegenüber anderen Spielern möglich.

8.2 Visualizer

Um Spiele im Nachhinein zu analysieren, haben wir die Spiele gelogt und abgespeichert. Diese können Mithilfe eines von uns dazu enteickelten Visiualisierungstools abgespielt werden werden. Dieser veranschaulicht mit Hilfe des Frameworks JavaFX das Spielgeschehen und ermöglicht ein Regulieren der Spielgeschwindigkeit. In Kombination mit den erstellten Logs können Spiele somit beliebig oft im Nachhinein

angesehen werden. Dies ermöglicht es zudem unbeaufsichtigt viele Testspiele zu absolvieren und diese gebündelt ohne Wartezeiten zu analyisieren. Das Logging der Spiele ist in der abgegebenen Version abgestellt.

8.3 Test-Server

Um nicht auf den offiziellen Server angewiesen zu sein und schnell testen zu können, haben wir einen eigenen Server zu Testzwecken entwickelt. Dieser hat zwei Versionen mit leicht unterschiedlichem Verhalten. Die Testversion mit bis zu drei leichten Bots als Gegnern und die Turnierversion mit den normalen Regeln. Der Testserver beendet das Spiel nicht, wenn nur noch ein Spieler übrig ist, damit wir sehen wie wir sterben, falls wir gewonnen haben.

8.4 GUI

Zur visuellen Darstellung der einzelnen Algorithmen verwenden wir eine GUI. Diese zeigt die einzeln berechneten Werte visuell anhand einer Farbe und einer Opacity auf freien Feldern an. Belegte Felder werden mit Rot für gegnerische Spieler und Grün für unseren eingefärbt und erhalten keine Opacity.
Die Opacity der einzelnen Felder ist dabei abhängig vom Algorithmus und entspricht dem Wert zwischen minimal und maximal, wobei maximal eine Opacity von 1.0 entspricht. Die Felder mit Incentives werden Cyan und die Felder mit Risiken gelb gefärbt.
Für die Darstellung von GLOBAL werden Felder mit Risiko von 1-3 mit gelb dargestellt und Incentives von 1-0 mit Cyan.
Der angezeigte Algorithmus kann während des laufenden Spiels gewechselt werden, aktualisiert sich aber erst mit dem nächsten Zug. Somit ist ein Wechseln des Algorithmus nach einem Spiel nicht mehr möglich.

9 Codequality

9.1 CI/CD

Um die Code-Qualität möglichst hoch zu halten haben wir von Beginn an auf CI/CD-Pipelines gesetzt. Hierzu wurden in Gitlab die Jobs “test” und “checkstyle” hinzugefügt(s. Abbildung).

Bei jedem Push auf einen Branch wurden diese Pipelines automatisch getriggert und haben im Falle eines Fehlers das Mergen als problematisch markiert.

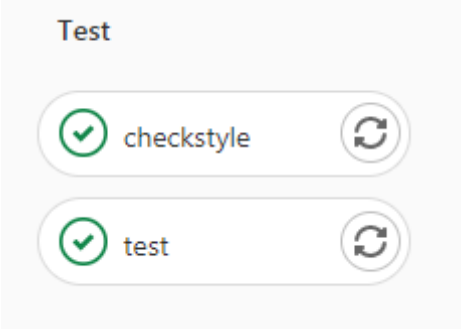

Figure 4: Automatische Jobs innerhalb der Gitlab Pipelines.

Durch zusätzliche Reviews in Form von Merge-Approvals auf einem Merge-Request konnte sichergestellt werden, dass jeder Code zusätzlich zu den automatisierten Tests auch von mindestens einer Person überprüft wurde.

9.2 Testen

Bei unseren Tests haben wir hauptsächlich auf Unit-Tests, sowie Property-Based-Tests gesetzt. Dazu haben wir die Frameworks JUnit und jqwik genutzt. Erweitert wurde unser Aufbau mithilfe der WhiteBox von Mockito.
Unser Ziel lag bei einer Test-Abdeckung von mehr als 50 Prozent.Da wir zu spät begonnen haben zu testen, sind Fehler durch Refactors entstanden, weshalb die Entwicklung teils durch intensivere Fehlersuche beeinträchtigt wurde.

9.3 Checkstyle

Neben dem Teststyle haben wir das Tool Checkstyle [CHE] genutzt, um unsere Code-Conventions zu kontrollieren. Unsere Conventions zur Einrückung, Klammerung und Leerzeichensetzung sind aus anderen gemeinsamen Projekte übernommen, und dienen der Lesbarkeit des Codes. Daneben verlangt die Convention, dass mehrzeilige öffentliche Methoden mit JavaDocs dokumentiert werden, um ein Mindestmaß an Dokumentation sicherzustellen.
Des Weiteren haben wir eine Überprüfung eingeführt, dass bei `switch`-Statements jeder `case` mit einem `break` oder `\\fallthrough` beendet wird und immer ein `default`-Fall vorhanden sein muss, um Fallthrough-Fehlern vorzubeugen.

9.4 Lombok

Wir haben uns zu Beginn des Projekts dazu entschieden, Lombok von Project Lombok [LOM] zu nutzen, um einfache Konstruktoren, Logger und Getter- und

Settermethoden zu generieren, damit die Klassen ohne viele Boilerplate-Methoden übersichtlich bleiben. Trotzdem sind gewohnheitsmäßig Getter und Setter geschrieben worden, die nicht alle mit Refactorings entfernt wurden.

Contents

References

[CHE] *Checkstyle*. URL: `https://checkstyle.sourceforge.io/` (visited on 01/17/2021).

[CPA19] John Song. *A Comparsion of Pathfinding Algorithms*. 2019. URL: `https://www.youtube.com/watch?v=GC-nBgi9r0U` (visited on 01/17/2021).

[IFC] informatiCup. *spe_ed*. URL: `https://github.com/informatiCup/InformatiCup2021/blob/master/spe_ed.pdf` (visited on 01/17/2021).

[LOM] *Project Lombok*. URL: `https://projectlombok.org/` (visited on 01/17/2021).

[MSA16] Felix Crevier, Sebastien Dubois, and Sebastien Levy. *Mutliplayer Snake AI*. 2016. URL: `http://sds-dubois.github.io/downloads/snake_ai.pdf` (visited on 01/17/2021).

[RTC] Rosettacode.org. *A* search algorithm*. URL: `https://rosettacode.org/wiki/A*_search_algorithm#Java` (visited on 01/17/2021).