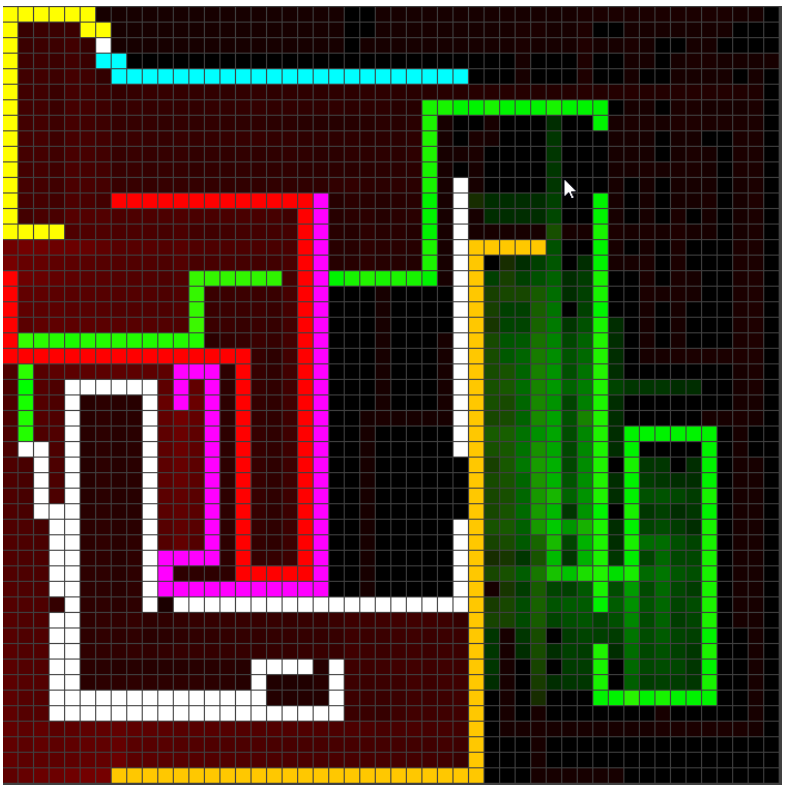
InformatiCup 2021 - Team Valhalla

Link to Github Repository

Bennet Martens, Fenno Boomgaarden

Januar 2021



Inhaltsverzeichnis

1 Einführung

Im Rahmen des InformatiCups 2021 hatte unser Team die Aufgabe, ein Programm zu entwickeln, welches ohne fremde Einwirkung das Spiel spe_ed spielen kann. In dieser Ausarbeitung werden wir unseren Lösungsweg, unsere Softwarearchitektur und unsere Algorithmen näher beleuchten. Am Ende werden wir unsere verschiedenen Ansätze vergleichen und so mit einer Analyse unserer finalen Abgabe schließen.

2 Lösungsweg

Das Team entschied sich zunächst dazu, mit einem Deep Learning Ansatz zu beginnen, weil alle Teammitglieder diesen Ansatz am interessantesten fanden. Er bot für uns eine gute Gelegenheit, erste Erfahrungen mit Deep Learning sammeln zu können. Uns war bewusst, dass eine algorithmische Lösung wahrscheinlich wesentlich bessere Ergebnisse erzielen würde, deshalb hielten wir uns diese Option offen, wenn unser Deep Learning Ansatz stagnieren würde. Wir begannen zunächst mit einer eigenen Implementierung des Spiels, der Websocket-Schnittstelle und mit einer graphischen Oberfläche. Dann entschieden wir uns, die Java-Implementierung der NEAT-Bibliothek (NeuroEvolution of Augmenting Topologies[1]) zu verwenden, da uns dieser Ansatz sehr vielversprechend schien. Wir schrieben eine Trainer-Klasse, die eine NEAT-Population darauf trainierte, das Spiel zu spielen und experimentierten mit den Parametern. Das trainierte Netz konnte dabei grundlegende Kompetenzen des Spiels lernen, wie z.B. belegten Feldern auszuweichen. Dennoch führten einfache Situationen, wie beispielsweise Sackgassen, oft zu Problemen. Daher entschieden wir uns angesichts der geringen verbleibenden Zeit für eine algorithmische Lösung.

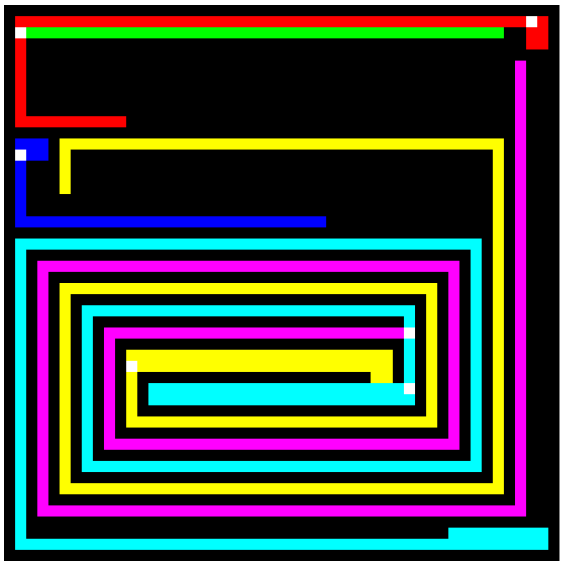


Abbildung 1: Höchstes erreichtes Spiel-Niveau mit dem NEAT-Ansatz

Die Grundlage unserer algorithmischen Ansätze ist ein Algorithmus zur schnellen Gebietsanalyse, der aus einem früheren Projekt eines Teammitglieds übernommen und abgewandelt wurde. Damit wurden die ersten algorithmischen Ansätze Siegfried und Brunhilde (angelehnt an der nordischen Mythologie) erarbeitet. Siegfried verwendet die einfache Strategie, immer in die Richtung zu fahren, in der die Gebietsanalyse das größte Gebiet gefunden hatte. Brunhilde war der Versuch, eine Implementierung der Mini-Max-Suche mit Alpha-Beta-Pruning zu verwenden. Die Mini-Max-Suche war, angesichts des großen Verzweigungsfaktors der Spielzüge, nur mit Bezug auf den nächstgelegenen Gegner möglich. Wir verwarfen diese Idee und entschieden, eine neue Baumsuche zu entwickeln, die die Züge der Gegner nicht beachten würde und so wesentlich tiefer rechnen könnte. So entstand der dritte algorithmische Ansatz, den wir Odin nannten, der bei hoher Suchtiefe bereits sehr erfolgreich über Spuren springen konnte. Als nächstes versuchten wir eine State-Machine zu benutzen, die für verschiedene Situationen verschiedene Strategien verwenden konnte. Diesen Ansatz verwarfen wir, als wir die Monte-Carlo-Baumsuche entdeckten. Wir implementierten diese mit unserer Gebietsanalyse als vierten Ansatz, Thor, und experimentierten mit den Parametern. Thor stellte sich in dieser abgewandelter Form der Monte-Carlo-Baumsuche als eine sehr gute Lösung heraus, die gezielt Sprünge und Angriffe plant und die meisten Spiele gegen unsere anderen Ansätze gewinnt.

3 Auswahl der Programmiersprache

Die Auswahl der Programmiersprache ist ein wichtiger Teil der Problemlösung, denn Leistung und Optimierung sind große Aspekte wenn man auf dem Server ein Spiel gewinnen möchte, weil wir vor unserem Zug 10 Züge in die Zukunft schauen wollen, damit unser Algorithmus den anderen immer einen Schritt voraus ist. Auch der Arbeitsaufwand wurde bei unserer Entscheidung mit einbezogen und aufgrund der im Team vorhandenen Kenntissen haben wir beschlossen in Java zu entwickeln.
Auch wenn Java im Punkt Leistung und Effizienz beim Lösen von mathematischen Formeln nicht mit maschinennahen Sprachen wie C mithalten kann, so hat Java doch die Vorzüge, dass man sich wegen der Garbage Collection nicht mit Speicherverwaltung beschäftigen muss. Außerdem waren ein Großteil der Rechenfunktionalität, die wir für die Lösung brauchten, bereits in Java implementiert.

4 Software-Architektur

Unsere Software-Architektur besteht aus unseren Spielklassen, die eine eigenen Implementierung des Spiels enthalten, unserer Server-Schnittstelle und unseren KIs.

4.1 Player und PlayerState

Die PlayerState-Klasse kann vom JSON-Parser erzeugt werden und enthält den Spielerzustand, wie er im JSON vom Server aufgebaut ist. Da wegen der Parser-Funktionalität alle Eigenschaften darin public sein müssen, kapseln wir den PlayerState in einer weiteren Player-Klasse. Die Player-Klasse enthält ein PlayerState-Objekt und bietet über Getter und Setter Zugriff auf dessen Eigenschaften. Außerdem bietet sie einige nützliche Funktionen, wie den Abstand von zwei Spielern zu berechnen.

4.2 Game und GameState

Wie bei der Player-Klasse kapselt die Game-Klasse ein GameState-Objekt, welches den gesamten Spielzustand enthält (mitsamt Liste der PlayerStates) und ebenfalls direkt aus JSON erzeugt werden kann. Die Game-Klasse führt seine eigene Liste von Player-Objekten, die die PlayerState-Objekte aus dem GameState kapseln. Außerdem enthält die Game-Klasse eine Offline-Implementierung des Spiels, die mit Game.create, Game.turn und Game.performMove verwendet werden kann, und eine Möglichkeit Spiele zu klonen. Dadurch lassen sich mit der Methode Game.variant Varianten des Spiels erzeugen, die unabhängig voneinander weitergespielt werden können. Desweiteren ist es möglich, mit Game.isDeadlyMove einschätzen zu lassen, ob für einen bestimmten Spieler ein bestimmter Spielzug direkt tödlich wäre.

4.3 Webbridge

Die Webbridge ist für ein- und ausgehende Nachrichten vom und zum Server zuständig. Sie verpackt den zuletzt gesendeten Spielzustand des Servers in einem GameState Objekt und sendet den gewählten Zug weiter an den WebSocketListener.

4.4 WebSocketListener

Der WebSocketListener ist eine eventbasierte Klasse, welche Fehlercodes verarbeitet (z.B. Status 427 - Already connected to server), gibt vom Server versendete Pakete an die Webbridge weiter und verschickt den von unserem Algorithmus ausgewählten Spielzug an den Server.

4.5 TimeSync

Von dieser Klasse wird für jede Verbindung eine Instanz erstellt, damit wir unsere Deadlines einhalten können. Sie ist dafür zuständig unsere Uhren mit dem Server zu synchronisieren und verwendet die Zeit-API des Servers.

4.6 Stage

Die Stage ist ein GUI, was unsere Spiele visualisieren kann. Sie kann sowohl selbst ein Spiel starten als auch ein bereits (z.B. auf dem Server) laufendes Spiel visualisieren. Dabei wird in grün auch eine Visualisierung der Thor-Baumsuche angezeigt, durch die

wir sehen, wieso sich der Algorithmus zu einem Zug entschieden hat und, in rot, mit welchen Zügen der Gegner unsere KI rechnet. Die Stage erbt von einem JFrame[3] und lässt sich mit Stage.setGame und Stage.repaint von außen steuern.

4.7 AreaFinder und Connector

Der Area-Finder verbindet den Connector-Algorithmus, den wir später näher erläutern werden, mit der Game-Klasse. Mit AreaFinder.setGame lässt sich festlegen, auf welchem Spiel gearbeitet werden soll. Mit AreaFinder.findAreas werden Connector-Objekte erzeugt, die voneinander getrennte Gebiete ausfindig machen. Mit AreaFinder.getAreaAt lässt sich zu einer Spielposition die Größe des damit verbundenen Gebietes ermitteln. Mit AreaFinder.getMaxAreaAround lässt sich von einer Position aus das größte angrenzende Gebiet ermitteln, auch wenn an der Position selbst bereits ein belegtes Feld ist (zum Beispiel bei einem Spieler). Die Komplexität von AreaFinder.findAreas liegt in $O(n \cdot log(n))$, wobei n die Anzahl an Feldern der Spielwelt ist.

4.8 MCTSNode

Die MCTSNode Klasse beschreibt einen Knoten der Monte Carlo Baumsuche[2]. Ein Knoten kann mit MCTSNode.expand seine Folgeknoten erzeugen, mit MCTSNode.addSimulation eine Bewertung zum Baum hinzufügen, mit MCTSNode.getConfidence die Vertiefungspriorität eines Knotens bestimmen und mit selectNode den Knoten unter den Kindknoten, der als nächstes berechnet werden soll, auswählen. Außerdem lässt sich der Pfad zum Knoten mit MCTS.getPath ausgeben. Damit der Suchbaum im nächsten Zug weiterverwendet werden kann, sodass langfristige Planung möglich ist, lässt sich der Baum mit MCTS.resetN normalisieren, sodass alle bisherigen Iterationen auf eine einzige reduziert werden. Damit bleibt ein Plan anpassungsfähig, sodass neue Berechnungen unter all den Alten nicht untergehen.

4.9 Algorithmische KIs

Zu guter Letzt haben wir in unserer Achitektur noch unsere algorithmischen KIs. Sie sind die Klassen, mit denen im Spiel alles steht oder fällt und sind natürlich das Herz unseres Programms. Jede von ihnen erbt von AlgorithmicAI, einer abstrakten Klasse, wodurch wir jeden Algorithmus einfach laden und auch ohne viel Aufwand Neue erstellen können. AlgorithmicAI.decide muss von der erbenden KI implementiert werden und gibt ein GameMove zurück. Die Klasse enthält ein Game-Objekt, das sich über einen Setter setzen lässt und eine playerId als Integer, die gemeinsam den Bezug zum Spiel definieren. Außerdem werden einige nützliche Analyse-Funktionen angeboten, insbesondere die Möglichkeit einen AreaFinder zu benutzen und ein einfacher Zufalls-Bot in AlgorithmicAI.randomMove. Der Zufalls-Bot wählt einen zufälligen Zug aus, der nicht sofort tödlich wäre.

5 Erklärung der Algorithmen

5.1 Gebietsanalyse

Die Grundlage unserer Lösung ist ein Algorithmus, der Gebiete in der Spielwelt identifizieren kann, unsere Gebietsanalyse. Jedem Feld wird dabei das Gebiet zugeordnet, zu dem das Feld gehört. Als Gebiet eines Feldes bezeichnen wir die Vereinigung aller Felder, die von dem betrachteten Feld aus ohne Sprünge erreichbar sind. Die Komplexität unseres Algorithmus liegt dabei in $O(n \cdot log(n))$, wobei n die Anzahl der betrachteten Felder ist. Der Algorithmus basiert auf eine Connector-Klasse, die aus einem anderen Projekt eines Teammitglieds wiederverwendet wurde.

Die Connector-Klasse ermöglicht es, in einem Graphen alle disjunkten Teilgraphen zu finden. Zunächst ordnet man jedem Knoten des Graphen ein Connector-Objekt zu. Jeder Connector repräsentiert einen Teilgraphen und kann einen Meta-Connector haben, sodass sich Connectoren zu einem Baum verbinden lassen. Zwei Connectoren lassen sich miteinander über Connector.merge verbinden. Wenn beide Connectoren einen Meta-Connector haben, wird das Connector.merge rekursiv auf die beiden Meta-Connectoren weitergeleitet. Hat nur einer der beiden Connectoren einen Meta-Connector, so bekommt der andere denselben Meta-Connector, sodass die beiden nun darüber verbunden sind. Haben beide Connectoren keinen Meta-Connector, so erzeugen sie einen gemeinsamen, der ihre Vereinigung repräsentiert. Sind die beiden Connectoren identisch, so kann abgebrochen werden. So kann einfach ein Mal über alle bestehenden Verbindungen von Knoten iteriert werden sodass die beiden den Knoten zugeordneten Connectoren mit Connector.merge verbunden werden. Danach repräsentiert der oberste Meta-Connector eines Knotens den zugehörigen disjunkten Teilgraphen.

In unserem Fall bekommt jedes freie Feld einen Connector, der mit den Connectoren der benachbarten freien Felder verbunden wird. Dann repräsentiert der oberste Meta-Connector eines Feldes sein Gebiet. Um die Größe des Gebietes ermitteln zu können, erweitern wir die Connectoren um einen Blätter-Zähler, der die Anzahl der Blattknoten im Baum zählt und auf seine Meta-Knoten übergibt. So ist die Blattzahl des obersten Meta-Knoten der Flächeninhalt seines Gebietes. Dieser Algorithmus muss allerdings nach jeder Spielweltänderung erneut durchgeführt werden, was bei großen Spielwelten noch immer relativ teuer sein kann. Dennoch wären andere Algorithmen, wie z.B. eine rekursive Nachbarschaftssuche wesentlich teurer. Die Gebietsgrößen lassen sich sehr gut als Spielevalutationen einsetzen und machen eine Sackgassenvermeidung trivial.

5.2 Odin

Odin ist eine AlgorithmicAI, die eine einfache Baumsuche und eine Evaluation auf Basis der Gebietsanalyse verwendet. Die Baumsuche verwendet die Game.variant Funktion um damit alle möglichen Folgezüge evaluieren zu können. Dabei wird der bisher beste Zug gespeichert und eine vertiefende Suche nur dann durchgeführt, wenn der letzte Zug ähnlich gut war, wie der bisher beste Zug. Das lässt sich über den Exploration-Wert setzen. Odin verfügt über zwei wichtige Evaluationen, die den AreaFinder benutzen: EvaluationMethod.area und EvaluationMethod.area_div_enemies. EvaluationMethod.area ist der An-

teil der eigenen Fläche an der in der gesamten Welt noch verfügbaren freien Fläche. Evaluationmethod.area_div_enemies ist die eigene Fläche im Verhältnis zur Summe aller Flächen der Gegner, was aggressiveres Spiel fördert.

5.3 Thor

Thor erweitert Odin um eine Abwandlung der Monte Carlo Baumsuche [2]. Die Monte Carlo Baumsuche im Original verwendet Zufallsspiele, um eine Spielsituation zu evaluieren. In einem Zufallsspiel ziehen alle Spieler so lange zufällige Züge, bis das Spiel beendet wird. In Spielen, in denen Spielerzüge nur selten tödlich sind, lässt sich damit über sehr viele Iterationen die Qualität eines Zuges sehr gut abschätzen. Die Monte Carlo Baumsuche besteht aus vier Phasen: Selektion, Erweiterung, Simulation und Rückverfolgung. Jeder Knoten führt dabei die Anzahl der von ihm ausgehenden Simulationen und die Anzahl der davon gewonnen Spiele, sodass sich daraus die Gewinnwahrscheinlichkeit des Knotens ergibt. Zunächst wird in der Selektion der nächste zu betrachtende Blattknoten im bisherigen Suchbaum ausgewählt. Diese Auswahl versucht die UCT-Funktion (**U**pper **C**onfidence bound 1 applied to **T**rees) zu maximieren, die eine Heuristik für die besten als nächstes zu betrachtenden Züge darstellt. Ist ein Blattknoten gewählt, so wird er auf seine Folgezüge erweitert. Dann wird von dem Blattknoten ausgehend ein Zufallsspiel gespielt (Simulation) und das Ergebnis an die Elternknoten weitergereicht (Rückverfolgung) sodass alle berührten Knoten im Baum eine Aktualisierung ihrer Gewinnwahrscheinlichkeiten erhalten. Diese vier Schritte lassen sich in beliebig vielen Iterationen wiederholen, wobei nach jeder Iteration etwas genauere Gewinnwahrscheinlichkeiten in den Folgeknoten des Wurzelknotens entstehen. Die KI muss dann nur noch den Knoten mit der höchsten Gewinnwahrscheilichkeit wählen. Wichtig dabei ist eine Balancierung der Suchbreite zur Suchtiefe. Dafür lässt sich ein Exploration-Faktor in der UCT-Funktion setzen. Ein guter Wert dafür hat sich aus der Literatur als $\sqrt{2}$ entnehmen.

Wir haben dieses Vorgehen implementiert und es mit unserem Odin verglichen. Weil die Zufallsspiele wegen der großen Zahl tödlicher Zugmöglichkeiten des Spiels meist nur sehr wenige Runden andauerten, haben wir in den Zufallsspielen die sofort tödlichen Züge ausgeschlossen. Dennoch verfangen sich die Zufallsspieler schnell in Sackgassen, sodass eine wirklich gute Evaluation mit Zufallsspielen in diesem Spiel nicht möglich ist. So verlor unsere Monte Carlo Baumsuche die meisten Spiele gegen unseren Odin, der durch sein gutes Gebietsverständnis überlegen war. Wir entschieden daher, die EvaluationMethod.area Evaluation von Odin mit der Baumstruktur der Monte Carlo Baumsuche zu kombinieren. Die Züge der Gegner werden dabei durch die Zufalls-KI repräsentiert. Diese werden allerdings bei jeder Iteration durch den Baum vollständig neu ausgewertet, sodass jeder Baumzweig nach ausreichend vielen Iterationen mit vielen der möglichen Gegnerzüge rechnet. Eine sehr tiefe Planung ist damit allerdings nur möglich, wenn kein Gegner in der Nähe ist. Wir haben außerdem beschlossen, den Suchbaum über mehrere Spielrunden zu erhalten, um langfristiges Planen zu fördern. So wird immer nur die Wurzel des Baumes auf einen der Kindknoten gesetzt und wenn ein langfristiger Plan ohne Gegnereinmischung möglich ist, kann sich dieser auf über 20 Züge in die Zukunft ausbilden. Insbesondere Befreiungspläne werden damit stark gefördert, sodass die KI

aus sehr brenzligen Situationen mit mehreren Sprüngen leicht entkommen kann. Wir haben außerdem festgestellt, dass mit unserer Evaluation die besten Ergebnisse mit einem UTC-Exploration Wert von 0 erzielt werden. Damit wird der Planungseffekt stark verstärkt was allerdings zu einer starken Ausdünnung des Suchbaumes führt. Das ist allerdings kein Problem, da anscheinend ein dünner, tiefer Suchbaum in diesem Spiel wegen der vielen tödlichen Züge sinnvoller ist, als ein breiter, flacher. Ein starker Vorteil dieser abgewandelten Baumsuche ist die adaptive Rechenzeit. Wir können solange Iterationen hinzufügen, bis die Deadline vom Server erreicht ist. Mit der bisherigen Baumsuche von Odin war das nicht möglich, da sie rekursiv den gesamten Suchbaum bis zu einer festgelegten Tiefe durchsucht.

6 Bewertung und Analyse

Unser wichtigstes Bewertungskriterium ist die Performance gegen unsere bisher besten KIs. Wir haben einen Performance-Test geschrieben, der über Nacht tausende Spiele mit verschiedenen KIs spielen und auswerten kann. Spielt ein Thor gegen fünf Odins der Tiefe 2, so erreicht Thor durchschnittlich den 2. Platz mit einer Gewinnwahrscheinlichkeit von 50 Prozent. Ein weiteres Kriterium waren Spiele, die wir gegen andere Teams unserer Universität spielen konnten. In einem inoffiziellen Tunier, was wir gegen drei Teams unserer Uni durchgeführt haben, hat Odin zwei von drei Spielen gewonnen und wurde bei einem Spiel Zweiter. Da Thor gegenüber Odin überlegen ist, erwarten wir eine Steigerung dieser Leistung. Wir haben außerdem hunderte Spiele unserer KIs angeschaut und qualitativ bewertet. Odin bringt sich häufig in Situationen, in denen Gegner sie mit Leichtigkeit abschneiden können. Thor hat dieses Problem nicht mehr, man kann deutlich sehen wie die KI immer den momentan sichersten Ort in der Karte anstrebt, was der offenste Ort mit den wenigesten Gegnern in der Nähe zu sein scheint. Wir vermuten dass dieser Effekt dadurch entsteht, dass von den Gegnern Zufallszüge mitgespielt werden. Odin hatte die Gegner einfach immer geradeaus fahren lassen. Thor versucht ihnen deswegen eher aus dem Weg zu gehen und spielt relativ passiv. Wir haben auch versucht Thor mit der aggressiven Evalutation auszustatten, allerdings wurden dann die Varianten stark abhängig von den zufälligen Gegnerzügen, sodass dadurch keine langfristige Planung mehr möglich wahr. Wir bevorzugen daher die passive Variante. Mit der Visualisierung der Baumsuche war eine noch bessere qualitative Analyse möglich, sodass insbesondere der Vorteil einer niedrigen UTC-Exploration entdeckt werden konnte. Thor kann Fluchtwege über Duzende Spielzüge und mit mehreren Sprüngen sehen und an dem Plan festhalten, kann aber auch seinen Plan dynamisch anpassen sobald ein Gegner in die Nähe kommt. Eine starke Schwäche liegt allerdings noch im direkten Kampf gegen Gegner. Wenn es möglich ist, einen Gegner abzuschneiden, wird diese Variante meist nicht gespielt, da dadurch auch die eigene Fläche reduziert werden würde. Im direkten Kampf gegen Odin kam es auch manchmal dazu, dass Thor von Odin eingeschlossen wurde und dass beide Spieler direkt ineinander gefahren sind. Solche Blindheiten können in seltenen Fällen auftreten, aber meist findet Thor einen guten Fluchtweg bevor es überhaupt zu einem richtigen Kampf kommt. Häufig kommt es im Kampf auch dazu, dass der Gegner

durch sein zu aggressives Verhalten an einen ungünstigen Ort verbleiben muss, während Thor einen Weg zu einem guten Ort geplant hat Ein etwas schwerwiegenderes Problem ist, dass Thor sehr häufig Fläche verschwendet und unnötig beschleunigt, obwohl er eigentlich alleine ist und auf Zeit spielen muss. Wir vermuten, dass ein schneller Zug ein besseres Verhältnis aus guten und schlechten Zügen im Suchbaum hat, weil die Anzahl sofort tödlicher Züge sofort steigt und von tödlichen Zügen keine Unterbäume berechnet werden. Bei einem langsamen Zug sind mehr mittelmäßige und schlechte Unterbäume vorhanden, sodass der Durchschnittliche Wert der Evaluationen geringer ist als bei einem schnellen Zug. Ein anderer Grund könnte sein, dass die KI möglichst schnell an den besten Ort navigieren will und diesen dann durch sein Bremsverhalten schnell zerstört und dann gleich zum nächsten besten Ort navigiert. Die Probleme waren uns bereits einige Wochen vor der Abgabe bekannt, aber trotz vieler Versuche konnten wir keine wirklich guten Lösungen dafür finden.

7 Zusätzliche Spielereien

Die .jar-Datei in unserer Abgabe lässt sich auch außerhalb des Containers starten, sodass Spiele in unserer Stage mit Visualisierung der Baumsuche auf 500 Iterationen gespielt werden können. Es ist auch möglich mit einem Klick auf die Leertaste die Kontrolle über den weißen Spieler zu übernehmen und ihn mit den Pfeiltasten zu steuern. Um diese Funktion zu nutzen, kann die Datei mit "java -jar TeamValhalla_Thor.jar stage guiäusgeführt werden. Dazu ist mindestens Java-8 erforderlich.

8 Fazit

Wir haben eine sehr leistungsstarke KI mit gutem Plan- und sehr interessantem Spielverhalten entwickeln können. Unser Exkurs in den Deep Learning Ansatz hat uns allerdings wertvolle Zeit gekostet, die wir am Ende noch hätten gebrauchen können. Mit stärkerem Kampfverhalten und effizienterer Raumnutzung wäre eine nahezu perfekte Lösung möglich gewesen. Dennoch waren die Erfahrungen, die wir im Rahmen des Projektes sammeln konnten, auch in Bezug auf den Deep Learning Ansatz, sehr wertvoll für die Weiterentwicklung unserer persönlichen Fähigkeiten, sodass wir diesen Schritt nicht bereuen. Wir sind gespannt, wie sich die anderen Teams gegen unsere Lösung behaupten und freuen uns auf die Ergebnisse.

Literatur

[1] Hunter Heidenreich, "NEAT: An Awesome Approach to NeuroEvolution",
https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

[2] James, Steven & Konidaris, George & Rosman, Benjamin, "An Analysis of Monte Carlo Tree Search.",
https://www.researchgate.net/publication/312172859_An_Analysis_of_Monte_Carlo_Tree_Searc

[3] Java Swing JFrame,
https://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html