Smart Computing

Foraschick, Dominic 222200429
Stettin, Tommy 221200375
Haase, Laurin Maximilian 217204840
Peters, Johannes 218205404

Sommersemester 2024

Inhaltsverzeichniss

1 Uml Diagramme

In diesem Abschnitt haben wir uns damit beschäfftigt unseren Bot in drei UML Diagrammen darzustellen. Wir haben uns dafür entschieden, ein Klassendiagramm, ein Sequenzdiagramm und ein Zustandsdiagramm zu modellieren. Wir haben uns für diese drei entschieden, da sie in ihrer Gesamtheit einen guten Überblick über die Funktionsweise unseres Bots geben.

1.1 Klassendiagramm

Die Entscheidung für ein Klassendiagramm basiert auf der guten Visualisierungsmöglichkeiten für die Beziehungen zwischen Klassen, Attributen und Methoden. Des Weiteren ist dieses Diagramm sehr gut geeignet, um aufzuzeigen, wo unsere Agentenfunktion verortet ist. Dies ist in diesem Fall mit einer Notiz im Diagramm vermerkt.

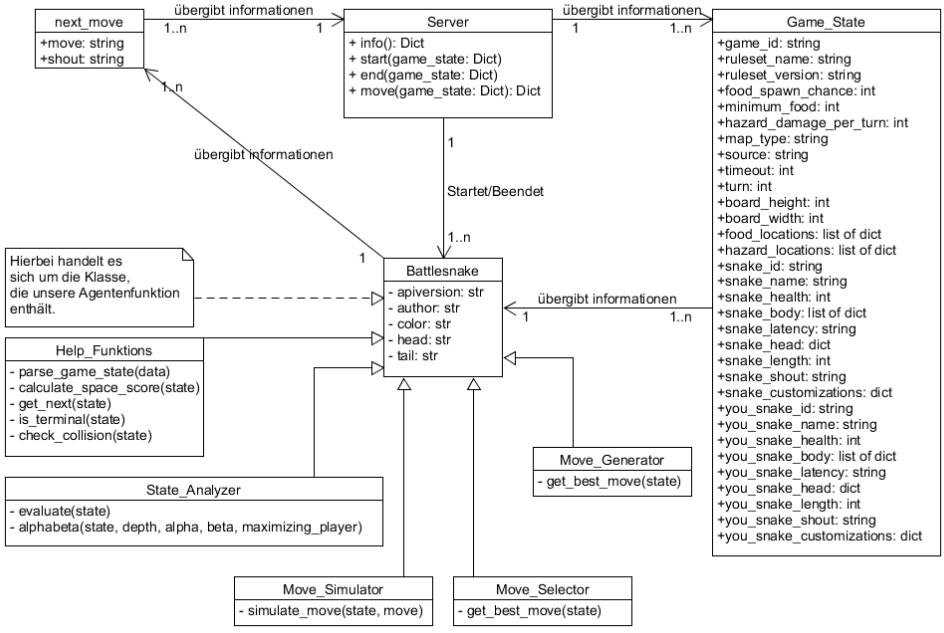

Figure 1: Klassendiagramm unseres Bots

1.2 Sequenzdiagramm

Für die Erstellung eines Sequenzdiagramms haben wir uns entschieden, um die Interaktionen zwischen verschiedenen Objekten, die in diesem Projekt einbegriffen sind, zu verdeutlichen.

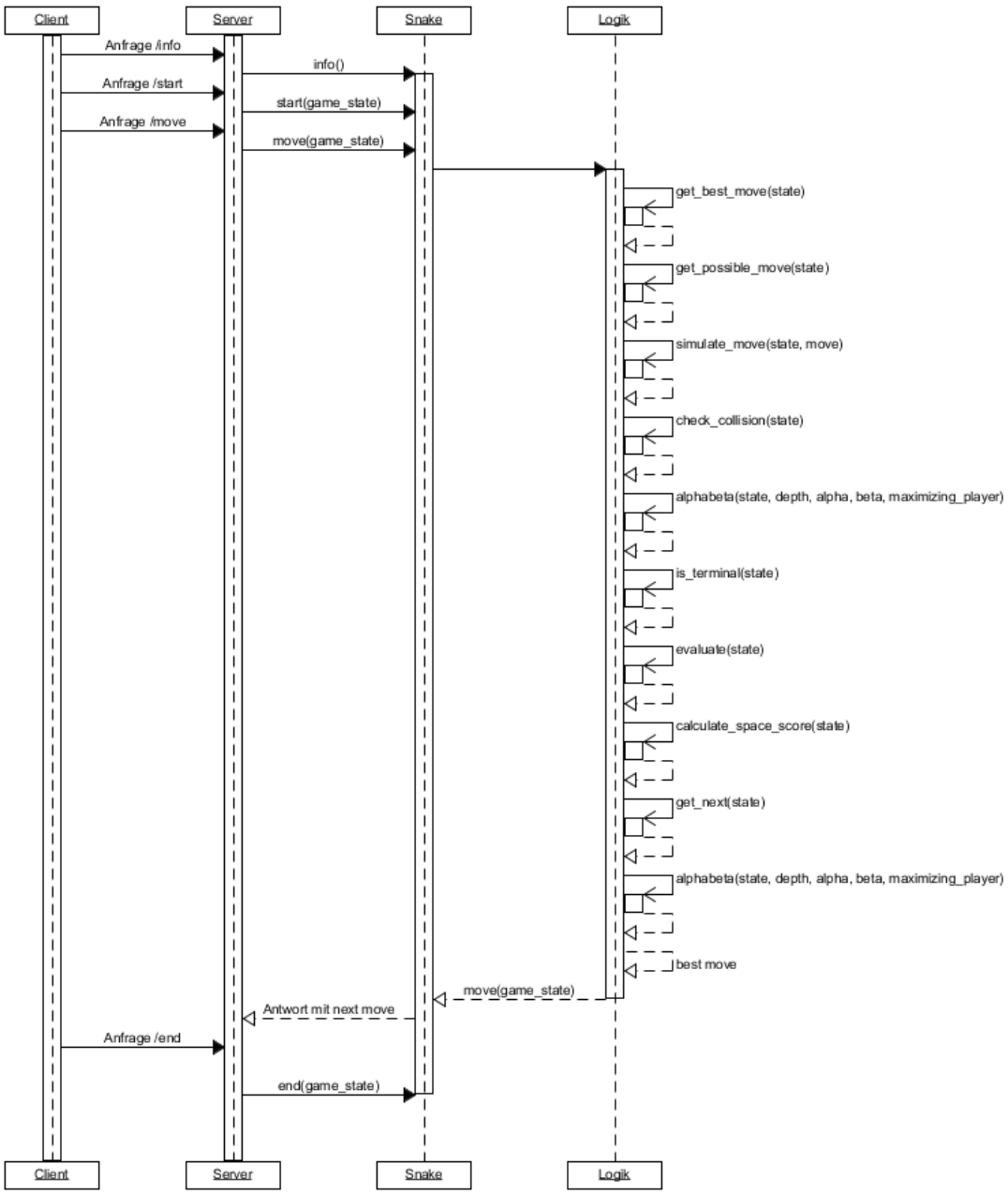

Figure 2: Sequenzdiagramm unseres Bots

1.3 Zustandsdiagramm

Für das sogenannte Zustandsdiagramm oder auch State-Chart haben wir uns als letztes Diagramm entschieden, da es sich sehr gut eignet, um die verschiedenen Phasen, die unser Bot während eines Spiels durchläuft, darzustellen.

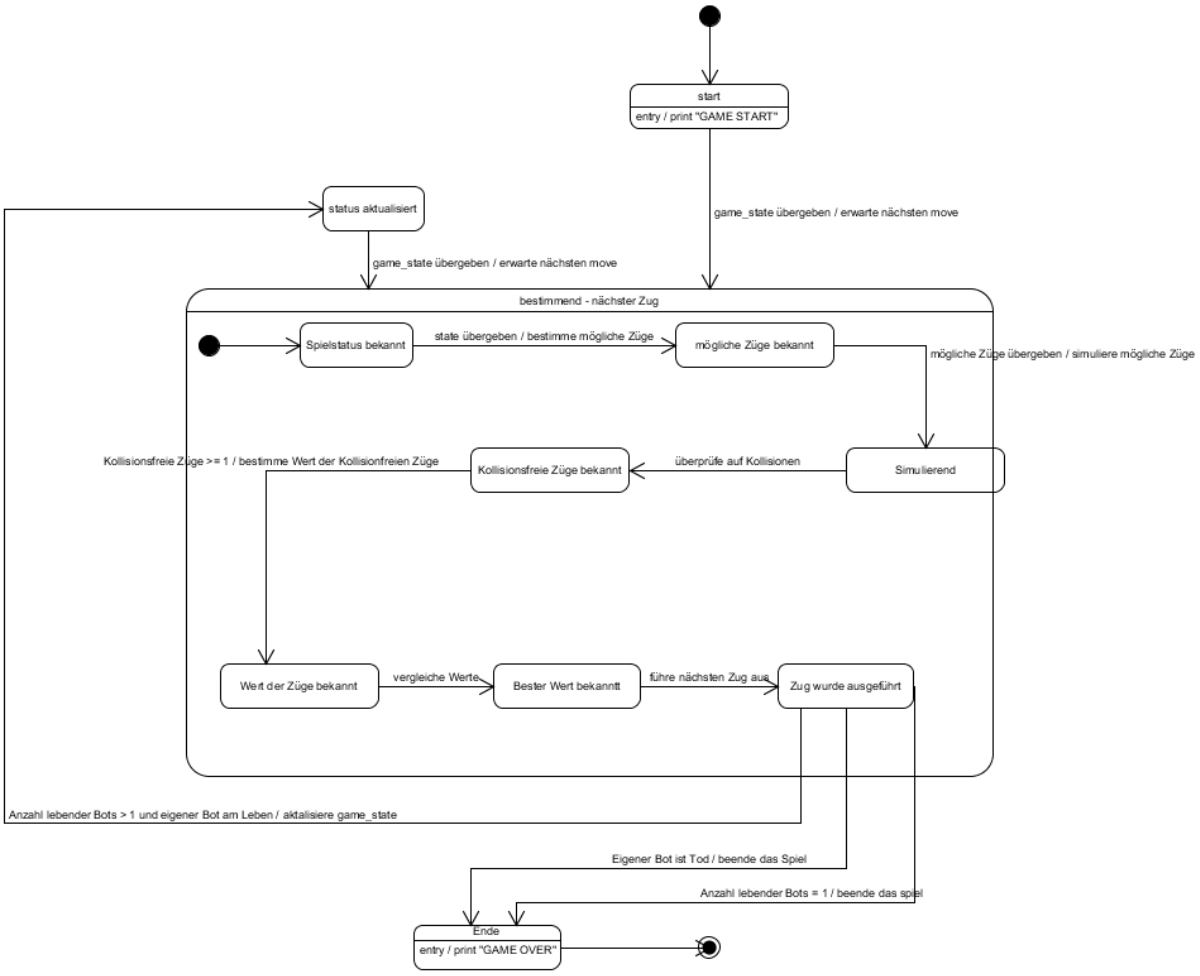

Figure 3: Zustandsdiagramm unseres Bots

2 Agent und Konzepte

2.1 Verwendeter Agententyp

Bei unserem Bot handelt es sich um einen rationalen Agenten. Das bedeutet, dass dieser Bot in der Lage ist seine Umgebung wahrzunehmen und im Rahmen dieser Umgebung bestimmte Aktionen ausführen kann. Das Ziel eines rationalen Agenten ist es so zu handeln, dass der Nutzen der ausgeführten Aktion des Agenten maximal ist. Im Kontext des Spiels "Battlesnake" bedeutet das soviel, wie den bestmöglichen Zug auszuwählen um am Ende des Spiels als letzter Bot übrig zu bleiben. Dies bedeutet jedoch nicht, dass die Aktionen absolut 'perfekt' sind, da die Aktionen anderer Bots nicht vorab bekannt sind.
Da diese Beschreibung auf einen großen Teil von Agenten zutreffen kann, wird unter den rationalen Agenten eine weitere Unterscheidungen vorgenommen. So können beispielsweise:

- Lernender Agent,
- Nutzbasierter Agent,
- Zustandsbehafteter Reflex-Agent oder Modellbasierter Reflex-Agent,
- Einfacher Reflex-Agent und
- Zielbasierter Agent

alle als rationale Agenten beschrieben werden. Unseren Bot würden wir als reflexbasierten Agenten mit Fähigkeit zur strategischen Planung verorten. Dass es sich hierbei um einen reflexbasierten Agenten handelt machen wir anhand von zwei Merkmalen fest.
Das erste ist hierbei die Reaktion auf die aktuelle Wahrnehmung. Unser Bot reagiert ausschließlich auf die Informationen, die er am Anfang eines jeden Zuges durch das Spiel bekommt und erfüllt dieses Kriterium somit vollumfänglich. Das zweite Kriterium auf das wir diese Einordnung stützen ist die direkte Reaktion des Bots. Das bedeutet, der Bot reagiert nur auf unmittelbare Ereignisse aus der Umwelt ohne eine Möglichkeit sich an bereits geschehene Ereignisse zu erinnern.
Die Fähigkeit zur strategischen Planung basiert auf der Bewertung des Ist-Zustands und der Simulation möglicher zukünftiger Zustände sowie der Bewertung dieser. Umgesetzt haben wir dies mithilfe des Minimax-Algorithmus mit angebundenem Alpha-Beta-Pruning. Die Entscheidung zur Verwendung dieser Methodik und deren Funktionsweise wird in den Abschnitten Verwendete Konzepte 2.3 und Begründung der Auswahl 2.4 erläutert.

2.2 Abgrenzung zu anderen Agententypen

Aufgrund der Ähnlichkeit unseres gewählten Agententyps zu anderen Typen wollen wir an dieser Stelle eine Abgrenzung zu einigen anderen Agenten vornehmen. Beginnen möchten wir dabei mit dem einfachen Reflex-Agenten. Dieser zeichnet sich beinahe gänzlich durch die selben Eigenschaften aus wie der von uns beschriebene Typ. Der Unterschied liegt hierbei in der Fähigkeit unseres Bots zur Simulation von möglichen zukünftigen Spielsituationen. Beide reagieren zwar nur auf den aktuellen Zustand der Umwelt, der von uns geschriebene Bot nimmt jedoch diesen Zustand um anhand der Daten die infragekommenden Aktionen von Gegnern zu simululieren und dies in die Entscheidungsfindung mit aufzunehmen. Der einfache reflexbasierte Agent hingegen hat diese Fähigkeit nicht und bewerten nur anhand des ist Zustandes.
Eine weitere Art von Agent von der eine Abgrenzung aufgrund der vorliegenden Ähnlichkeit wichtig ist, ist der Modellbasierte Reflex-Agent. Auch dieser trifft Enscheidungen anhand des aktuellen Zustandes, verwendet dabei jedoch auch noch ein internes Modell der Welt. Dieses dient dazu, sowohl bereits zurückliegende als auch zukünftige Zustände in die Entscheidungfindung mit aufzunehmen. Die Abgrenzung zu unserem Bot liegt hierbei eindeutig in der Fähigkeit historische Zustände mit einzubeziehen, da unser Bot diese nicht hat und ausschließlich in die Zukunft blickt.
Der letzte Agenttyp von dem wir eine Unterscheidung vornehmen wollen ist der Zielbasierte Agent. Dieser definiert sich dadurch, dass er spezifische Ziele verfolgt und das Erreichen besagter Ziele durch eine selbständige Analyse der Zustände anstrebt. Nun könnte man unseren auch in dieser Kategorie verorten, da er Ziele wie das Sammeln von Nahrung und das Überleben priorisiert. Der Unterschied liegt hierbei jedoch in der Entscheidungsfindung, ein Zielbasierter Agent ist in der Lage dazu selbständig eine Analyse durchzuführen. Der Bot für den wir uns entschieden haben nimmt diese Analyse nicht gänzlich selbständig und dynamisch vor, sondern anhand der Vorgaben die wir durch beispielsweise den Minimax-Algorithmus mit Alpha-Beta Pruning implementiert haben.

2.3 Verwendete Konzepte

Zur Optimierung unserer Snake haben wir uns wie bereits erwähnt entschieden, unseren Bot um die Methode des Minimax Algorithmus mit Alpha-Beta Pruning bzw. der Alpha-Beta Suche zu erweitern. Der größte Vorteil der durch diese Erweiterung gewährleistet wird, ist neben den Vorteilen die der Minimax Algorithmus an sich schon mitbringt, eine durch das Alpha-Beta Pruning ermöglichte eine deutliche Steigerung der Effizienz ohne, dass das Ergebnis dabei verändert wird.
Der Minimax-Algorithmus arbeitet dabei mit einem Maximierer und einem Minimierer. Normalerweise ist dieser Algorithmus für zweispieler Spiele ausgelegt, in unserem Falle haben wir jedoch die Gegner als Minimierer zusammengefasst. Der Algorithmus geht zur Bestimmung des besten Zugs nun so vor, dass er einen erzeugten Suchbaum bis zu einer bestimmten Tiefe an einem Pfad ent-

lang durchsucht. Der Maximierer also unser Bot sitzt hierbei immer in der Wurzel des Baums und danach wechseln sich Maximierer und Minimierer bis zur vorherbestimmten Tiefe ab. An diesem Punkt hat der Algorithmus dann die Blätter des Baums erreicht, die jeweils den Ausgang einer bestimmten Zugreihenfolge darstellen. Die Bewertung der Ausgänge wird in unserem Fall durch eine Funktion durchgeführt, die Faktoren wie Entfernung zu Essen oder Gefahren einbezieht und den Blättern anhand dieser Daten Werte zuweist. Nun wird je nach dem welcher der beiden Akteure grade am Zug ist entweder der maximale oder der minimale Wert rekursiv bis zur Wurzel zurück gegeben. Dieses Vorgehen wird nun wiederholt, bis jeder Pfad des Baumes durchsucht wurde. Der große Vorteil hierbei liegt darin, dass nun selbst wenn der Minimierer optimal handelt der gewählte Zug den besten Ausgang garantiert, auch wenn der Minimierer optimal handelt.
Da die Suche durch alle Pfade des Baumes durchaus Zeitintensiv sein kann haben wir uns wie bereits erwähnt dazu entschieden, den Minimax-Algorithmus um die Methode des Alpha-Bete Prunings zu erweitern. Dieser kann je nach Aufstellung des Baumes eine enorme Zeitreduktion beim Durchsuchen des Baumes gewährleisten. Dies wird erreicht, in dem bestimmte Abschnitte des Spielbaums gar nicht erst durchsucht werden müssen. Das Ausschließen basiert darauf, Varianten, welche von Anfang schon ein schlechteres Ergebniss lieferen würden als bisher untersuchte, gar nicht mehr zu prüfen. Um dies zu gewährleisten wird jedem Knoten im Baum ein Intervall alpha-beta zugeordnet. Alpha wird mit -unendlich initialisiert und stellt hierbei den größten bisher gefundenen Wert da, den ein Max-Knoten sicherstellen kann. Im gegensatz dazu wird beta mit +unendlich initialisiertden und stellt den kleinsten Wert dar, den ein Min-Knoten sicherstellen kann. Es wird nun so vorgegangen, dass zu erst durch Tiefensuche die linke seite des Baumes des Baumes abgearbeitet wird, um referenzwerte für alpha und beta zu erhalten. Diese werden nun übergeben. Der Max-Knoten übergibt nun den wert alpha an den nächsten Kindknoten. Wenn es in diesem Min-Knoten dazu kommt, dass beta einen Wert kleiner oder gleich alpha annimt kann die Suche in diesem Knoten abgebrochen werden. Die Begrundung dafür liegt an der Tatsachte, dass sich der Max-Knoten in keinem Fall mehr für diesen Zug entscheiden würde. So wird nun der ganze Baum durchsucht, bis der optimale Minimax-Wert gefunden ist.

2.4 Begründung der Auswahl

Abgesehen von den bereits gennannten Vorteilen 2.3, die unser Entscheidung den Minimax Algorithmus mit Alpha-Beta Pruning zu verwenden beeinflusst haben, haben wir zur Entscheidungsfindung eine auf ein kollektives Testen verschiedener Konzepte verständigt. Bei diesem Prozess hat sich schlussendlich der Minimax-Algorithmus mit Alpha-Beta Pruning als unsere beste Option durchgesetzt. Wie wir dabei vorgegangen sind und weshalb es die anderen Konzepte es nicht in die engere Auswahl geschafft haben wird in Abschnitt **??** und **??** beschrieben. Der Minimax-Algorithmus mit alpha-beta-pruning

überzeugte unter anderem, durch seine Annpassbarkeit im Laufe der Entwicklung. Ebenso zeigte sich schnell die Möglichkeit, weitreichendere strategische Folgen einzelner Züge einzuschätzen, und auch das Verhalten von Gegnern umfänglicher zu berücksichtigen, als unentbehrlich.

3 Überlegungen und weitere Strategien

Bevor wir uns für die jetzt implementierten Strategien entschieden hatten, haben wir Prototypen auf Grundlage einiger weiterer Ansätze au der Vorlesung entwickelt, mit dem Ziel den Vielversprechendsten zu wählen und auszubauen.
Dabei haben wir Überlegungen angestellt zur Umsetzug der dominanteb Strategie, sowie eines A*-Algorithmus. Beide Strategien

3.1 Überlegungen

3.1.1 Dominate Strategie

Spieler scheiden aus, wenn sie verhungern, oder mit Schlangen oder der Spielfeldgrenze kollidieren. Genügend vorausschauende Konkurrenten können diese Fälle in der Regel gut vermeiden. Schließlich können Schlangen auch eliminiert werden, indem eine größere Schlange ihnen den Kopf abbeißt, also mit diesem kollidiert. Letztere Option ist die effektivste um das Spielziel zu erreichen. Eine solche Kollision können wir selber herbeiführen und direkt beeinflussen durch unsere Zugwahl.
Dies ist der Kerngedanke der dominanten Strategie. Statt Sicherheit und Überleben als höchstes Ziel anzusehen, wollen wir jede Mäglichkeit wahrnehmen andere Schlangen auszuschalten. Beziehungsweise diese zu bedrohen um ihre Optionen einzuschränken.
Essen wurde hier geringer priorisiert, als die Nähe zu anderen Schlangen zu vermindern. Jedoch werden insbesondere kürzere Konkurrenten so auch von Essen abgedrängt. Der Prototyp welcher die dominante Strategie anwandte zeichnete sich durch sehr kurze, volatile Runden aus. Die verminderte Distanz zwischen Schlangen führte häufig und früh im Spielverlauf Kollisionen herbei. Durch dieses aggressive Verhalten steigt jedoch außerdem das Risiko für unseren Agenten. Kann die dominanten Schlange nicht einen im frühen Spielverlauf erlangten Längen-, oder Raumvorteil als Hebel benutzen, so ist der auf Konkurrenten ausgeübte Druck durch den Wettbewerb um Platz deutlich weniger gefährlich.
Dabei zeigt sich auch eine große Herausforderung für diesen Agenten, nämlich das Risiko für sich selbst angemessen gegen das für die Gegner abzuwägen. Insbesondere führte der Agent sich selbst häufig in riskante bis aussichtslose Situationen, in denen er selbst früh eliminiert wurde.

3.1.2 A*

A* war ein weiterer Ansatz, den wir anfangs verfolgt hatten. Dieser schien zunächst vielversprechend. Jedoch bereitete das Fehlen einer Definition für den

Zielbereich Schwierigkeiten.
Möglich wäre die Menge der Zustände, in denen das Spiel gewonnen ist, als Endzustände zusammenzufassen. Jedoch war es dann schwer eine Heuristik zu entwickeln, die einen angemessenen "Abstand" zu dieser Zustandsmenge beschreibt. Oder überhaupt diese Zustandsmenge sinnvoll abzugrenzen.
Die Beurteilung einzelner Züge gestaltete sich als schwer auf Grund der schieren VIelfalt möglicher Endzustände, der Anzahl von Zügen bis zu einem Endzustand, sowie der unklaren Eigenschaft "Abstand" zu dieser diversen Menge von Endzuständen.
Aber für direkte Wegfindung zeigte sich ein A*-Ansatz jedoch durchaus als geeignet. Um beispielsweise den besten Pfad zu Essen in einem bestimmten Spielzustand zu ermitteln. Außerdem ließ sich dieser Wegfindungsansatz leicht adaptieren. Z.B. zu dem Fall, in dem die Schlange sich in einem Passivmodus befindet. Dies sollte ein Default-Verhaltensmuster sein, in welchem die Schlange ihr eigenens Ende verfolgt und sich im Kreis bewegt, solange Gefahr und Hunger keine große Rolle spielen.
Die Umsetzung eines allunmfassenden A*-Ansatz scheiterte daran, eine geeignete zulässige Heuristik zu entwerfen welche Zuständen eine Distanz zu der großen, diversen Menge möglicher Endzustände zuweist, und Züge so beurteilt. Als Wegfindung zu einem festen, wenige Schritte entfernten Ziel konnte dieser Ansatz mit der Manhattendistanz als Heuristik umgesetzt werden.
Wie genau wir diese Heuristik implementiert haben wird im weieren Verlauf dieses Kapitels betrachtet.

3.2 Folgende strategische Entscheidungen

Aus unseren Prototypen stellte sich schnell der Minimax-Nutzende als der arbeitbarster heraus. Aus unseren anderen Vorgehen adaptierten wir die Manhatten-Metrik von einer reinen Wegfindungsheuristik für Essen zu einer weiterreichenden, welche nun außerdem den Abstand zu Konkurrenten bemisst und somit ein kritiches Maß für die Sicherheit von Spielzügen und Zuständen bietet.

3.2.1 Manhatten-Metrik

Eine Manhatten-Metrik ist ein Distanzmaß, welches die ganzzahlige Differenz zwischen zwei Koordinaten bemisst. Im Kontext des zweidimensionalen Battlesnake-Spielfeldes ist die Manhattendistanz zwischen 2 Punkten

$$A(x_a, x_b)^T, B(x_b, y_b)^T$$

ist die Manhattendistanz d gleich

$$d(A, B) = |x_b - x_a| + |y_b - y_a|$$

und bemisst damit die Anzahl von Schritten, bzw. Spielzügen welche zwei Punkte trennen.
Damit ist die Manhattendistanz ein geeignetes Maß für Battlesnake, deren Wert

in Bezug zu einem Ziel den Grad der Mittelbarkeit dieses Zieles ausdrückt. Ferner lässt sich dieses Maß auch direkt auf die diskreten Züge in Battlesnake übertragen: Essen an Punkt B ist von einer Schlange deren Kopf an Punkt A liegt in frühestens d(A,B) Zügen erreichbar. Diese Bewertung liefert also kritische Informationen zum vorausschauenden Handeln, um beispielsweise sicherere Pfade anzustreben, oder zu erkennen, wenn Essen unangeofchten zu erreichen ist.
Schließlich ist das Maß einfach und effizient zu berechnen, spiegelt gut den rundenbasierten Charakter des Spiels wider, und bietet eine intuitive Lösung Distanzen zu berechnen und Züge anhand entstehender Distanzen zu bewerten.

3.3 Begründung der Auswahl

A* war effektiv für Wegfindung, aber berücksichtigte nicht den weiterreichenden Kontext von Spielstrategie oder dem Verhalten von Gegnern.
Der dominanten Strategie, wenn auch effektiv unter bestimmten Voraussetzungen, fehlte es an Anpassbarkeit und strategischer Tiefe.
Für unsere Evaluationsfunktion, auf deren Grundlage das in Abschnitt 2.3 pruning geschieht, bedient sich zur Bemessung von Gefahr durch Konkurrenten und Nähe zu Essen einer Menhattenmetrik aus den oben aufgeführten Gründen.

4 Rundumblick Battlesnake

Das Spielfeld von Battlesnake ist ein Quadrat aus Zellen (11x11, 19x19 etc.), das sich als zweidimenstionales Array darstellen lassen kann. Jede Zelle ist leer, enthält ein Segment einer der Schlangen, oder Essen. Informationen über das Spielbrett, sowie den Inhalt der Zellen werden regelmäßig zu jedem Zug vom Spielserver übermittelt. Ferner bleiben die Dimensionen des Spielfeldes im Laufe einer Runde konstant.
Die Schlangen versuchen am Längsten zu überleben - nur der "last man standing" gewinnt. Also stehen alle Agenten von Anfang an im direkten Wettbewerb miteinander. Die bessere Platzierung eines Wettstreiters bedeutet zwingend den Abstieg eines Anderen.
Die Spielumgebung ist dynamich: die Schlangen verändern ihre Positionen mit jedem Zug. Schlangen scheiden aus, Essen wird gefressen oder spawnt neu.
Kollidiert eine Schlange mit den Grenzen des Spielbrettes, mit einem Segment einer anderen Schlange oder sich selbst so ist sie eliminiert. Ebenso werden Schlangen, welche 100 Züge in Folge kein Essen konsumieren durch Hunger eliminiert. Kollidieren zwei unterschiedlich lange Schlangen Kopf an Kopf, so scheidet die kürzere aus, während die längere überlebt. Essen stillt dabei Hunger immer vollständig, und verlängert die Schlange um ein Segment. Essen erscheint dabei im Laufe des Spiels auf auf verscheidenen Feldern.
Kommunikation oder Koordination mit mit den anderen Agenten ist effektiv nicht möglich, da über den Server der selbe Spielstatus jede Runde an jeden Agenten übermittelt wird, und es keine Kommunikationsmöglichkeiten darüber hinaus gibt. Der einzige Informationsaustausch ist das jede Runde an die Spieler übermittelte Spielfeld, und der von ihnen übermittelte Zug.
Allerdings entscheiden die Agenten mit vollständiger Information. Zu jedem Zeitpunkt ist allen Agenten der Zustand des Spiels vollkommen und gleichermaßen bekannt. Damit sind die Positionen und Lebenspunkte jeder Schlange sowie das verfügbare Essen eingeschlossen. Selbst Informationen darüber wie beispielsweise Essen generiert wird sind öffentlich verfügbar und stehen theoretisch allen Agenten zur Nutzung bereit.
Zwar ist das Wissen über die Spielumgebung jederzeit vollständig, ausgenommen des Verhaltens der anderen Agenten. Das Spiel kann auf den ersten Blick als deterministisch betrachtet werden. Jedoch vornehmlich unter der Annahme, dass jeder teilnehmende Agent bei jedem herrschenden Zustand genau einen möglichen Zug ermittelt, welchen er spielt. Sofern zumindest einer der Agenten in einem Fall keine eindeutige Entscheidung für einen Zug findet, und zufällig wählt, so ist das Spiel nichtdeterministisch. Außerdem entsteht weitere Variabilität durch die Zufälligkeit von Startpositionen. Damit ist Battlesnake, außerhalb hochkontrollierter Umstände, in denen solche Zufälligkeiten ausgeschlossen sind, nichtdeterministisch. Treten die selben Agenten mehrmals gegeneinander an, so hat außerhalb dieser Einschränkungen nicht zwingend jedes Spiel das selbe Ergebnis.
Die Vollständigkeit von Informationen gilt ferner nur für den aktuellen Zug. Entschiedungen anderer Agenten sind im Verlauf einer Runde im Vorhinein

nicht eindeutig bestimmbar. Somit nimmt der Grad der Infomationsvollkommenheit für jeden Agenten ab, je weiter ein betrachteter Zug in der Zukunft liegt.
Das Spiel schreitet über diskrete Runden fort. In jeder Runde treffen alle Agenten eine Entschiedung über ihren je nächsten Zug. Der folgende Zuständ, mit den gespielten Zügen, eventuellen Essens-spawns oder Disqualifikationen wird dann wieder an alle verbleibenden Agenten übermittelt. Die Spielregeln sorgen dafür, dass ein Folgezustand - mit gegebenem Vorgängerzutand und Zügen der Spieler - immer deterministisch ist. Die Folgen einer Handlung sind immer bekannt: jeder der Bewegungbefehle bewirkt immer exkat das Selbe. So kann das Gewinnen einer Runde Battlesnake als wohldefiniertes Problem angesehen werden, mit einem Startzustand, konstanten Schrittkosten von 1 für alle Züge (bzw. unendlich für Züge, welche die Schlange eliminieren). Die Menge von Zielzuständen enthält dann alle Zustände, in welchen nur noch die Spielerschlange am Leben ist. Ebenso kann ein Spiel als constraint-satisfaction-Problem verstehen werden, wobei Züge die in Kollisionen resultieren, oder 100 aufeinanderfolgende Züge ohne Essen können als constraits formuliert werden, edren Verletzung die Eliminierung aus dem Spiel widerspiegelt. Jede Schlange muss sich jede Runde bewegen, wächst wenn ihr Kopf mit Essen kollidiert, und verhungert wenn ihr Leben auf 0 sinkt. Diese Regeln, sowie Kollisionen werden serverseitig verarbeitet und bestimmten den Folgezustand.

5 URL

An diesem Punkt ist die URL vermerkt, unter der unsere Snake errreichbar ist. Zum erfolgreichen erreichen der URL muss jedoch das Replit Repository aktiviert werden. Das problem, weshalb es nicht dauerhaft erreichbar sein kann ist, dass ein dauerhaft aktives repository nur in der premium Version Replits enthalten ist.

https://4f9ce24b-0c07-4e1d-85f6-ddeede0b71dc-00-867b0rhcjyv9.riker.replit.dev/