Contents

1 SnakeKIs

1.1 Übersicht

Julia und ich haben mehrere KIs zum testen erstellt. Bis auf die letzte Abgabe war es auch immer zielich einfach welche KI unsere HorstAI wird, also die Abgabe KI. Zu unseren KIs gehören:

1. **AlphaBetaSnake**
   Diese KI verwendet ausschließlich unseren AlphaBetaAlgorithmus um den bestmöglichen nächsten Zug zu bestimmen. Diese KI haben wir am Ende auch als HorstAI verwendet, da sie im Turnier häufiger gewinnt als BrainMaster(die 2. KI, die für HorstAI zur Auswahl stand)

2. **BrainMaster**
   Die BrainMaster KI versucht eigentlich alles zu verwenden, was wir programmiert haben. Diese KI also bitte auch mit bewerten, da sie denke ich auch recht gut zu lesen ist. Wir haben hier alles in Methoden ausgelagert und in *nextDirection* nacheinander nach Wichtigkeit

sortiert aufgerufen. Eine genaue Beschreibung folgt in einem eigenen Kapitel.

3. **HorstProto**
   Diese KI war zum Zwischenspeichern einer früheren Version gedacht um im eigenen Turnier vergleichen zu können ob die KI besser geworden ist.

4. **NewBrain**
   Julia hat mit dieser KI einen Algorithmus ausprobiert, damit die Schlange möglichst lange am Leben bleibt. Leider sind hier wahrscheinlich noch ein paar Fehler im Code.

5. **NewBrainTest**
   Hier wurde der Code von Julia genommen und versucht dort noch die Algorithmen einzusetzen, die wir schon fertig ausprogrammiert hatten, aber es hat zum Ende hin leider nicht mehr gereicht die Fehler raus zu bekommen.

6. **NotMovingBrain**
   Diese KI läuft zur unteren rechten Ecke und läuft dort nur hin und her. Sollte sie irgendwie von dort weg gedrängt werden, versucht sie wieder diese Ecke zu erreichen. Man kann diese KI verwenden, wenn man testen möchte ob die eigene KI lange überlebt ohne Einwirkungen von Gegnern.

7. **SurvivalAI** Mit dieser KI haben wir uns überlegt, wie wir es schaffen können garantiert länger zu überleben als die gegnerische Schlange. Das Ziel dieser KI wurde dann also immer der gegnerischen Schlage zu folgen. Das funktioniert eigentlich schon recht gut. Die KI gewinnt sogar ein paar mal im Turnier. Das Problem hier ist aber noch, dass sie wirklich dicht hinter der gegnerischen Schlange bleibt und nur zur Seite ausweicht, wenn die gegnerische Schlange einen Apfel frisst. Das Problem, dass die Schlange der anderen dicht folgen soll, kann man ähnlich wie bei *NotMovingBrain* lösen, indem man einen boolean "inPosition" auf true setzt und ab dann nur noch den nächsten Zug abhängig vom Schlangenkörper des gegners wählt.

1.2 BrainMaster

Da die BrainMaster KI unsere 2. größere KI war (und bis auf die letzte Abgabe immer abgegeben wurde) sollte sie genauer betrachtet werden.

1.2.1 init

Wie in den meisten anderen unserer KI's folgt hier auch, in der *nextDirection*-Methode, der Aufruf der *init* Methode. Diese sollte nur im 2. Schritt lange brauchen. In dieser Methode wird alles was benötigt wird initialisiert, falls das noch nicht geschehen ist. Wir haben diese Methode auf 2 Schritte aufgeteilt, da wir hier auch einen kompletten HamiltonPath über das gesamte Spielfeld berechnen, den wir uns speichern. Durch Threading könnten wir das vielleicht zusammenführen zu einer Runde.

1.2.2 AlphaBeta-Pruning

Das erste was die KI berechnet ist auch mit AlphaBeta-Pruning eine Einschätzung aller 4 Richtungen und damit die Bewertung davon. Hier aber mit einer geringeren Tiefe als bei der *AlphaBetaSnake*, da wir hier noch mehr machen. Diese KI verwendet AlphaBeta-Pruning nur um eine direkte Gewinnmöglichkeit zu ermitteln. Das heißt wenn der *bestScore* höher als 9000 ist, hat unser Algorithmus eine Möglichkeit gefunden, wie unsere Schlange direkt gewinnen kann. Falls das der

Fall ist folgen wir diesem Weg ohne weitere Berechnungen. Die Bewertungen werden aber auch verwendet um sehr schlechte Wege auszuschließen. Danach gehen wir alle 4 Positionen durch und sagen unserem PathFinder für den kürzesten Weg auch, dass er diese Positionen möglichst ausschließen soll um zum Ziel zu kommen. Das geschießt über *minPathFinder.addBadPosition* Falls wir hier mehr als 2 schlechte Züge haben, vertrauen wir auch unserem Algorithmus und laufen direkt in die beste Richtung, die wir zur Verfügung haben.

1.2.3 wallDetection

Falls wir weder gewinnen noch verlieren können, versuchen wir uns Gewinnoptionen zu generieren. Mit der Funktion *wallNextToAppleDetection* überprüfen wir ob ein Apfel direkt neben einer Wand ist. Dies codieren wir als Binärzahl mit 4 Bits für die jeweilige Richtung wo eine Wand ist.

oben	rechts	unten	links
8	4	2	1

Damit wäre beispielsweise ein Apfel in der oberen linken Ecke als: **1001** codiert, also **8 + 1 = 9**

1.2.4 Falle für die gegnerische Schlange legen, wenn möglich

Wenn wir einen Apfel an der Wand haben, dann legen wir unter bestimmten Bedingungen eine Falle für die gegnerische Schlange. Diese Falle kann dann so aussehen:

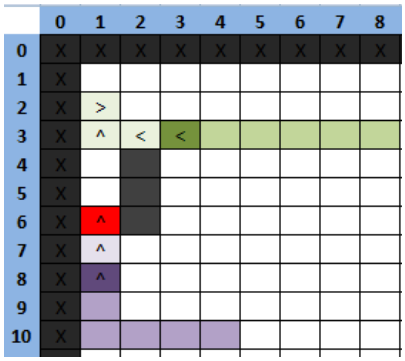

Figure 1: Falle an der linken Wand. Unsere Schlange ist grün, die gegnerische Lila

- dunkelgrün/dunkellila : Schlangenkopf
- grün/lila : Schlangenschwanz
- hellgrün/helllila : voraussichtlicher Weg
- dunkelgrau : Wand
- rot : Apfel
- <,v,,> : Richtung in die gelaufen wird

Mit diesem Bild werden auch die Bedingungen klar:

1. die gegnerische Schlange muss nach setzen der Wand schneller beim Apfel sein als wir.
2. unsere Schlange muss nach dem setzen der Wand schneller am *closePoint* sein, als die gegnerische Schlange. Das wäre auf diesem Bild der Punkt (3,1) *(Zeile,Spalte)*.
3. unsere Schlange muss lang genug sein um der gegnerischen Schlange keinen Ausweg mehr zu lassen.

Mit diesen Bedingungen kann man das ganze analog für alle 4 Seiten überprüfen und dabei auch Position, der zu setzenden Wand anpassen.

1.2.5 Schlange kürzen?

Als nächstes prüfen wir ob unsere Schlange eine Maximal-Länge überschritten hat. Ist dies der Fall versuchen wir durch ein Portal die Schlange wieder auf eine gute Länge zu bringen. Das machen wir, da wir so auch etwas den Faktor verringern wie leicht wir uns einschließen können. Hier müsste man dann auch noch das CutTail-Feature mit berücksichtigen.

1.2.6 Äpfel essen!

Nun folgt der wohl wichtigste Punkt beim Snake-Spiel: Äpfel essen und Punkte machen. Wir nutzen hier allerdings auch unseren Binärcode für die Position des Apfels. Wenn dieser an einer Wand liegt essen wir diesen zur Zeit nicht. Hier sollte man in Zukunft dann noch Ausnahmen einbauen. Damit die Schlange wirklich nur dann den Apfel nicht frisst, wenn sie dadurch verlieren würde. Wenn beim Apfel keine Wand ist, gehen wir auch nur zum Apfel. wenn wir näher dran sind (Portale werden hier mit berücksichtigt). Alternativ wird hier dann auch noch überprüft ob wir die Schlange tauschen können bevor die gegnerische Schlange den Apfel frisst.

1.2.7 Kein Apfel? dann eben eine Wand!

Haben wir bisher kein Ziel gefunden versuchen wir ein WallFeature-Item zu bekommen, damit wir eine Falle bauen können. Das machen wir aber nur, wenn wir das Item noch nicht haben.

1.2.8 Immernoch kein Ziel in Sicht? Alternativen?

Wenn wir bisher immernoch nichts als Ziel festgelegt haben, dann versuchen wir auf Zeit zu spielen und bewegen uns zu unserem alternativ Ziel. Diese sind:

- Kopfposition der gegnerischen Schlange
- mittig am unteren Rand
- mittig am linken Rand
- mittig am rechten Rand

Um erkennen zu können ob dieser Weg sinnvoll ist (falls wir uns eingeschlossen haben, ist das nicht sinnvoll), testen wir direkt ob wir zu 2 verschiedenen dieser alternativ Ziele kommmen können. Wenn wir zu beiden können, bewegen wir uns zum derzeitig festgelegten Ziel (*currentAltTarget)*)

1.2.9 Was? eingeschlossen? Wir müssen hier wieder raus!

Damit wären wir dann auch beim letzten Ausweg, falls wir uns eingeschlossen haben. Hier berechnen wir zuerst ob wir durch ein Portal raus können oder durch vertauschen des Schlangenkopfs mit dem Schlangenschwanz. Falls das nicht geht berechnen wir den LongestPath zu der Stelle, an der wir als erstes wieder raus kommen können. An welcher Stelle das ist bestimmen wir durch den kürzesten Pfad. Wenn wir vom Schlangenkopf zum Schlangenschwanz den kürzesten Pfad berechnen und es diesen nicht gibt, ist dieses Segment außerhalb des eingeschlossenen Bereichs. Dann arbeiten wir uns Element für Element zum Kopf bis es diesen kürzesten Pfad gibt. Dieses Segment ist dann der Punkt an dem wir als erstes den Bereich verlassen können. Wir nehmen dann also den kürzesten Weg her und vergrößern ihn solange bis wir nichts mehr erweitern können. Den gesamten Weg speichern wir in einem Path Stack, den wir dann ablaufen bis wir draußen sind.
Falls wir damit keine Lösung finden, versuchen wir unseren HamiltonPath über das gesamte Spielfeld zu verwenden.

1.2.10 Jetzt ist es auch egal. Mach irgendwas

Hat bisher nichts funktioniert, dann sind wir wahrscheinlich gerade dabei zu verlieren. Dann ist es nun auch egal was wir machen, also beweg dich mit randomMove.

2 Algorithmen

2.1 HamiltonPath/LongestPath

Unser Algorithmus für den HamiltonPath oder LongestPath funktioniert wie folgt: Angenommen wir haben das folgende Spielfeld

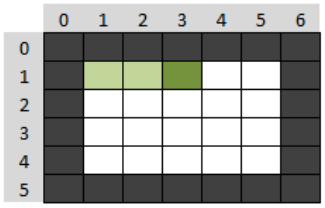

Figure 2: Test Spielfeld

Wenn wir dann bis auf den Schlangenschwanz und den Kopf alle Schlangensegmente als *unbegehbar* bezeichen und dann den kürzesten Weg von Kopf zum Schwanz berechnen entsteht der folgende Weg:

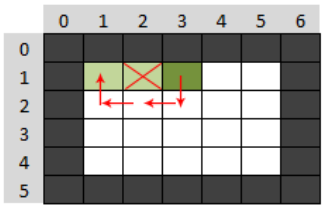

Figure 3: kürzester Weg von Kopf zum Schlangenschwanz

Unser Algorithmus versucht das nun für jedes Schlangensegment durch, vom Schlangenschwanz angefangen, indem es immer dieses eine Element nicht als *unbegehbar* bezeichnet. Irgendwann wird es dann den kürzesten Weg zu diesem Segment geben und das ist dann auch wie oben beschrieben unser Punkt, an dem unsere Schlange aus einer eingeschlossenen Situation als erstes wieder raus kann.
Mit diesem kürzesten Pfad beginnt unser Algorithmus dann nun erst richtig. Wir gehen den Pfad Richtung für Richtung ab und schauen ob wir ihn ausbauen können.

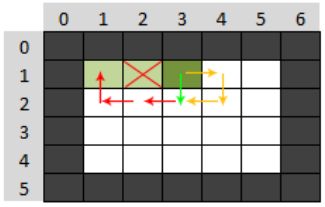

Figure 4: Erste Richtung nach unten ausgebaut

Zum Beispiel die erste *Direction.DOWN* Richtung können wir ersetzen durch einen längeren weg nach rechts,unten und dann links. Damit bleibt der Ausgangspunkt der gleiche. Das können wir aber nur tun, wenn die 2 Felder daneben noch *begehbar* sind. Wir sehen schon, dass der daraus resultierende Weg wieder mit der ersten *Direction.DOWN* Richtung wieder nach dem gleichen Muster erweitert werden kann. Daraus entsteht nun folgender Weg:

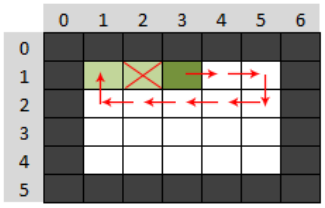

Figure 5: Nochmals Die Richtung nach unten ausgebaut

Gehen wir den Weg wieder durch, können wir die erste Richtung nach links erweitern.

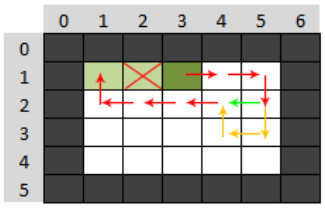

Figure 6: Richtungsausbau Direction.LEFT

Führen wir das nun weiter aus und erweitern jede Direction, wenn möglich. Dann erhalten wir am Ende unseren LongestPath/HamiltonPath.

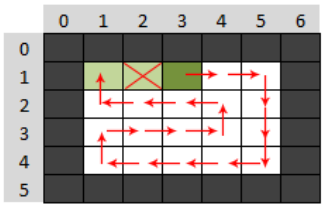

Figure 7: Test Spielfeld

Wir führen diese Erweiterungen so lange aus, bis sich der Pfad nichtmehr ändert bei einem Durchlauf. So können wir sicher gehen, dass unser Pfad maximal ist.

3 Fazit

3.1 Was lief besonders gut?

3.1.1 Wegsuche

Bei der Entwicklung der KI lief vor allem der A*-Algorithmus ziemlich schnell. Das lag vielleicht auch daran, dass ich ihn schon ein paar mal programmiert habe. Ich konnte diesen dann auch noch ziemlich runter kürzen. Das hinzufügen der Portale war anfangs auch nicht so einfach, aber lief dann auch erstaunlich gut, so dass der A*-Algorithmus auch Wege durch Portale berücksichtigt. Dafür hatte ich dann auch kurzzeitig einen Test geschrieben, den ich aber auskommentiert habe, da er Funktionen benötigt, die ich mir für diesen Zweck in den gegebenen Klassen selbt hinzufügen musste.

3.1.2 Tests

Wenn wir schon bei Tests sind. Diese zu implementieren war lief auch sehr gut. Wir konnten durch diese Tests für uns bestätigen, dass unsere Algorithmen genau das tun, was sie sollen. Da der HamiltonPath eine Zeit lang nicht funktioniert hat, konnten wir diesen in einer Art Testdriven Development korrigieren und dann auch direkt validieren.

3.1.3 Algorithmen verstehen

Auch das Verständnis für den Algorithmus vom HamiltonPath hatte man schnell verstanden und diesen konnten wir dann auch schnell grob umsetzen und mit den Tests dann fertig implementieren.

3.1.4 Eigenes Feature einbauen

Erstaunlich schnell konnten wir unsere beiden Features einbauen. Wir konnten uns direkt zurecht finden im Code und an den nötigen Stellen die Änderungen einfügen. Im Grunde war das auch nicht besonders schwer, weil wir ja bereits eine Vorlage mit dem WallFeature hatten. Das einzige Schönheitsproblem war hier dann noch der Tausch von Kopf und Schwanz der Schlange, der etwas aussah wie ein Wurmloch/Portal. Das konnten wir aber auch sehr schnell berichtigen.

3.2 Was war schwierig?

3.2.1 AlphaBeta Umsetzung

Den Algorithmus für AlphaBeta-Pruning konnte man ja recht schnell von Wikipedia runter kopieren. Aber die nötigen Stellen durch unsere Spielumgebung zu ändern, stellte sich als sehr schwierig heraus. Hier mussten wir vorallem mit den Referenzen aufpassen. Damit nach dem verändern des Spielfelds nicht das tatsächliche Spielfeld verändert wird, sondern eine Kopie. Genauso mit den Schlangen. Am Ende hat das aber dann alles irgendwie geklappt. Nur bricht der Algorithmus nun ab, sobald unsere Schlange den Apfel erreicht hat und evaluiert dann das Spielfeld. Hier müsste man sich noch überlegen, was die Schlange nach dem Apfel erreichen soll und dafür in der eval Funktion Punkte geben.

3.2.2 Feature einbau

Wir haben uns am Anfang ja alle Features überlegt. Ich finde inzwischen, dass manche davon nur schwer in die KI mit eingebaut werden konnten. Zum Beispiel das WallFeature. Hier war unser Problem immer, dass wir uns nie richtig überlegen konnten, wie wir das nun verwenden können, so dass es nicht einfach nur random irgendwo eine Wand setzt. Ich hätte das auch gerne in den AlphaBeta Algorithmus eingebaut, aber alle Positionen durchtesten hätte dann die Laufzeit doch erheblich gesteigert und mir ist auch kein Weg eingefallen, wie wir das geschickt dort einbauen hätten können.
Aber auch das Geschwindigkeits-Feature oder das OpenField Feature konnten wir nicht richtig einbauen, da uns dort am Anfang die richtige Idee gefehlt hat und wir mit anderen Sachen hinterher waren, die wir als wichtiger eingestuft hatten.

3.2.3 Zeit Faktor

Mit der Zeit hatten wir es manchmal auch nicht einfach. Manche Features haben schon viel Überlegungszeit benötigt um sich einen guten Einsatz in der KI zu überlegen. Dazu kam dann auch noch das Threading...

3.2.4 Threading

Mit dem Threading hatten wir so unsere Probleme. Julia und ich hatten mit Threading noch nicht viel zutun gehabt und hatten bei dem Thema erstmal so unsere Probleme. Wir konnten uns erst nicht so richtig vorstellen, wo wir bei unserer KI das Threading einbauen können, da alles ja so ein bisschen aufeinander aufbaut. Erst gegen Ende hatte ich dann noch eine Idee für

so eine KI. Es hätte aber zu viel Zeit gebraucht, diese ordentlich umzusetzen. Die Idee war jede unserer Berechnungen in einem eigenen Thread auszuführen und am Ende zu bewerten, was in der aktuellen Situation das beste ist. Ähnlich wie bei AlphaBeta die eval-Funktion.

3.3 Worauf bist du stolz?

Besonders stolz bin ich auf den HamiltonPath-Algorithmus. Ich hatte am Anfang so meine Zweifel, dass das alles funktioniert. Die Tests haben mir dann aber auch diese Zweifel genommen.
Aber auch auf den AlphaBeta-Algorithmus bin ich inzwischen stolz. Inzwischen funktioniert er ja schon recht in Ordnung. Hier ist aber auch das größte Erweiterungspotenzial vorhanden.

3.4 Wer hat welchen Code geschrieben?

Den Großteil des Codes haben wir zusammen geschrieben. Also Julia an ihrem Laptop und ich an meinem. Ich habe dann noch viel an dem HamiltonPath und AlphaBeta-Algorithmus gearbeitet während Julia an der NewBrain KI geschrieben hat. Die Tests haben wir beide zusammen angefangen, Julia hat dann noch weitere Tests geschrieben von Situationen, die uns während der Laufzeit aufgefallen sind. Die Code Dokumentation/Code Refactoring haben wir auch zusammen angefangen und dann teilweise aufgeteilt. So hat Julia den PathFinding Algorithmus und dessen Hilfsklassen dokumentiert während ich AlphaBeta und den HamiltonPath dokumentiert habe.

3.5 Wer hat welche Ideen gehabt?

Ideen hatten wir eigentlich viele, aber auch sehr viele verworfen. Die Idee mit dem AlphaBeta Agenten hatte ich durch die Vorlesung Künstliche Intelligenz bei Prof. Zell. Während Julia den Algorithmus für NewBrain gefunden hat. Dabei war dann auch der Algorithmus für den HamiltonPath etwas erklärt. Diesen habe ich dann ausprobiert. Ansonsten waren unter meinen Ideen die Falle für die andere Schlange, wenn der Apfel an einer Wand liegt und die Idee, dass sich die Schlange um das Schlangentausch-Feature wickelt (das wurde ja dann leider abgeschwächt :))

3.6 Was hättet ihr anders gemacht?

Wir haben wahrscheinlich zu viel Zeit damit verbracht Dinge auszuprobieren, die unserer KI nicht viel bringen. Wir hätten uns wahrscheinlich einen besseren Plan am Anfang auslegen sollen und nicht bunt drauf los programmieren. Damit hätten wir den AlphaBeta-Algorithmus vielleicht schon früher auf dem jetzigen Stand gehabt und hätten diesen dann nur durch die neuen Features erweitern müssen. Auch das GIT hat nicht so ganz funktioniert wie es sollte, da es auf Julias Netbook nicht lief und sie nur aus dem GIT runterladen konnte, aber nichts pushen/committen.

3.7 Wie lief die Zusammenarbeit

Wir haben uns 1-2Mal in der Zeit getroffen und dort so viel wie möglich fertig programmiert. Oft habe ich dann noch selbt ein bisschen was zuhause ausprobiert und das je nach Erfolg ins GIT mit gepusht.

3.8 Was fehlt noch?

Unsere KI ist noch lange nicht perfekt, wie man wahrscheinlich auch an der Positionierung im Turnier sieht. Ich würde mich nun hauptsächlich auf den AlphaBeta-Agenten konzentrieren und diesem beibringen, was das Ziel ist, wenn der Apfel weg ist und was getan werden soll, wenn der gegner näher am Apfel ist. Das sind alles Dinge, die in die eval-Funktion gehören. Zusätzlich müsste man schauen ob man die Struktur vom AlphaBeta-Algorithmus noch etwas ändern kann indem man nicht über eine undomove-Funktion arbeitet sondern direkt auf einer Kopie des Spielfelds und für undo einfach wieder das alte Spielfeld verwendet.
Wir müssten uns auch mehr Strategien überlegen, wie wir die Features nutzen wie WallFeature,SpeedUp,OpenField usw.

3.9 Was sind Stärken und Schwächen unserer KI?

3.9.1 BrainMaster

Die BrainMaster KI hat denke ich noch die Schwächen, dass sie noch nicht aggressiv genug die andere Schlange angreift und angriffe nur sehr schlecht abwehren kann. Sie läuft quasi bereitwillig in die Fallen rein. Ihre Stärken dagegen sind denke ich, dass sie inzwischen schon recht lange überleben kann, da sie sich immer wieder verkürzt wenn sie zu lang geworden ist und durch den HamiltonPath-Algorithmus kann sie auch aus eingeschlossenen Situationen hin und wieder herauskommen.

3.9.2 HorstAI(AlphaBetaSnake)

Diese KI ist denke ich stark gegen andere KIs, da sie immer versucht direkt zu gewinnen. Das gute an diesem Algorithmus ist, dass man schon einen Schutz vorm einschließen mit dabei hat je tiefer man sucht desto besser ist dieser Schutz. Das liegt daran, dass wir in der eval Funktion den Tod unserer Schlange schon als sehr schlecht bewerten und wir durch den Algorithmus diese Situationen möglichst verhindern.
Zu ihren Schwächen zählen denke ich die fehlenden *Parameter* in der eval Funktion um auch ohne Apfel immer eine gute Entscheidung treffen zu können. Auch die Rechenzeit ist bei dieser KI noch nicht ganz optimal. Diese Punkte kann man aber alle beheben indem man für ersteres die eval Funktion ausbaut und verbessert. Für die Performance findet sich für den Alpha-Beta Algorithmus auf Wikipedia ganz gute Ansätze. Dabei würde die Zugsortierung schon einen ordentlichen Schub raus holen.
`https://de.wikipedia.org/wiki/Alpha-Beta-Suche`