

Bachelorarbeit
in der Angewandten Informatik
Nr. AI-2020-BA-012

Implementierung eines Computergegners für Chamäleon Schach nach heuristischem Ansatz

Hannes Dröse

Abgabedatum: 31.08.2020

Prof. Dr. Ines-Kerstin Rossak
Prof. Dr. Steffen Avemarg

Kurzfassung

Diese Bachelor Arbeit beschäftigt sich mit der Implementierung eines Computergegners für das Brettspiel “Chamäleon Schach”. Die Umsetzung erfolgt mithilfe von klassischen Algorithmen nach dem sog. heuristischen Ansatz. Besondere Herausforderung dabei ist, dass das Spiel nicht nur zu zweit, sondern auch zu dritt oder zu viert gespielt werden kann. Für Zwei-Spieler-Spiele gibt es einen klaren Standard-Algorithmus, den MiniMax mit Alpha-Beta Pruning. Ab drei Spielern gibt es verschiedene Algorithmen, die geeignet sein könnten. Deshalb werden in dieser Arbeit verschiedene solcher Algorithmen implementiert und miteinander verglichen. Die Algorithmen sind: der Max^N in seiner Standard-Implementierung und mit verschiedenen Pruning-Verfahren, der Hypermax und die paranoide Version des MiniMax mit Alpha-Beta Pruning.

Abstract

This bachelor thesis is about the implementation of a computer opponent for the board game “chameleon chess”. Classic Algorithms were used to solve this problem, they follow the so-called heuristic approach. The most challenging part is that the game can be played by two, three or four players. For two player games there is a standard algorithm – the MiniMax with Alpha Beta pruning. However, for three players or more there are many different algorithms that could be used. Some of them were implemented and their performance was compared. The algorithms were: Max^N in its standard implementation and using different pruning methods, the hypermax and the paranoid version of MiniMax with Alpha Beta pruning.

Inhaltsverzeichnis

Tabellenverzeichnis	V
Abbildungsverzeichnis	VI
1 Ziel dieser Arbeit	1
2 Chamäleon Schach	2
Entstehung	2
Spielprinzip	3
3 Spieltheorie	5
Allgemein	5
Einordnung von "Chamäleon Schach"	6
Spielanalyse	6
Spielbaum (engl. game tree)	6
Equilibrium-Punkte/Sicherheitsniveau	7
Perfekte Spielweise	10
Komplexität von Brettspielen	10
Anzahl verschiedener Spielverläufe	10
Anzahl verschiedener Spielsituationen	11
4 Heuristischer Ansatz	13
Prinzip	13
Heuristische Bewertungsfunktion	13
Beschränktes Durchsuchen des Spielbaums	13
Pruning	13
Iterative Deepening	14
Algorithmen	14
Minimax mit Alpha-Beta Pruning	14
Max ^N	16
Hypermax	19
5 Implementierung	21
Bewertungsfunktion	21
Algorithmen	22
Allgemein	22
Max ^N	23
Max ^N IS	23
Paranoid	24
Hypermax	24
Session-Framework	25
Algorithmus-Factory	25
Session	26
6 Auswertung	28
Bewertungsfunktionen	28
Paranoid - normiert oder nicht	29
Hypermax - normiert oder nicht	30
Bester Algorithmus für Chamäleon Schach	31
7 Fazit und Ausblick	35
Quellen	VII

Anhang

Spielanleitung für Chamäleon Schach	VIII
Spielziel	VIII
Vorbereitung	VIII
Spielverlauf	X
Spielende	XI
Züge der Spielfiguren	XI
Spezialfall	XII
Source Code	XIII
Selbstständigkeitserklärung	XIV

Tabellenverzeichnis

3.1	Die Komplexität von “Chamäleon Schach”	11
5.1	Wertigkeiten der Schach-Rollen anhand ihrer Bewegungsweite	21
5.2	Die zu vergleichenden Bewertungsfunktionen	22
6.1	Vergleich der Bewertungsfunktionen mit fester Tiefe	28
6.2	Vergleich der Bewertungsfunktionen mit Rollengewichtung mit fester Tiefe . .	29
6.3	Vergleich des Paranoid mit und ohne Normierung mit fester Suchtiefe . . .	29
6.4	Vergleich des Paranoid mit und ohne Normierung mit fester Rechenzeit . .	30
6.5	Vergleich des Hypermax mit und ohne Normierung mit fester Tiefe	30
6.6	Vergleich des Hypermax mit und ohne Normierung mit fester Rechenzeit . .	30
6.7	Vergleich der vier Algorithmen mit fester Suchtiefe	31
6.8	Vergleich der vier Algorithmen mit fester Rechenzeit	32
6.9	Max ^N IS vs Hypermax mit fester Rechenzeit von 1.000ms	34
6.10	Max ^N IS vs Paranoid mit fester Rechenzeit von 1.000ms	34
6.11	Hypermax vs Paranoid mit fester Rechenzeit von 1.000ms	34

Abbildungsverzeichnis

2.1	Der Holzkasten der ersten Version von “Chamäleon”	2
2.2	Mitten im Spiel von “Chamäleon”	2
2.3	Das Spielbrett von “Chamäleon Schach”	3
2.4	Die Spielfiguren von Spieler Rot mit ihren Farbe-Rolle-Zuordnungen	3
2.5	Die Startaufstellung der Figuren bei einem Vier-Spieler-Spiel	4
2.6	Das Schrumpfen des Spielbretts	4
3.1	Spielbaum von TicTacToe für die ersten beiden Züge	7
3.2	Ausschnitt vom Spielbaum von TicTacToe gegen Ende des Spieles mit Auszahlungen	8
3.3	Durchschleifen der Auszahlungen bei Zügen ohne Entscheidung	9
3.4	Auszahlungen für alle Knoten in diesem Ausschnitt des Spielbaums	9
3.5	Spielbaum von TicTacToe mit Auszahlungen	10
4.1	Alpha-Beta Pruning	15
4.2	Rückwärtsauflösung im paranoiden Ansatz	16
4.3	Arbeitsweise des Max ^N	16
4.4	Shallow Pruning. $\max\Sigma = 10$	17
4.5	Deep Pruning liefert falsche Ergebnisse. $\max\Sigma = 10$	18
4.6	Unterschiede zwischen Max ^N und Hypermax	19
7.1	Das Logo von Chamäleon Schach	VIII
7.2	Das farbenfrohe Spielbrett	IX
7.3	Das Spielbrett mit den Spielern an den Seiten	IX
7.4	Das Spielbrett mit den Figuren in Startaufstellung	IX
7.5	Die Figuren eines Spielers mit den jeweiligen Farbe-Rolle Zuordnungen	X
7.6	Einengung des Spielbretts	X
7.7	Züge eines Springers	XI
7.8	Züge eines Läufers	XI
7.9	Züge eines Turms	XII
7.10	Züge einer Dame	XII

1 Ziel dieser Arbeit

Ziel dieser Arbeit ist es, einen Computergegner für das Brettspiel “Chamäleon Schach” zu entwickeln. Zur Zeit wird eine digitale Version des Spieles erarbeitet. Es entsteht eine App für mobile Endgeräte. Die Umsetzung erfolgt mit dem JavaScript Framework React Native. Die komplette App ist also in JavaScript implementiert. Daher soll der Computergegner optimalerweise ebenfalls in JavaScript geschrieben werden.

Eine besondere Herausforderung ist, dass die App auf dem mobilen Endgerät ausgeführt wird, ohne Nutzung eines Servers. Das heißt, dass komplizierte Berechnungen nicht in einem Cloud Service ausgelagert werden können. Alle Berechnungen müssen auf dem Endgerät selbst, mit den entsprechend beschränkten Ressourcen, durchführbar sein. Im Design-Prozess der App ist festgestellt worden, dass es besser aussieht, wenn der Computer nicht “sofort” seinen Zug ausführt, sondern sich eine kleine “Bedenkzeit” lässt. Dabei hat sich ein Zeitraum von einer Sekunde als optimal erwiesen. Die Berechnungen können also ohne Probleme bis zu einer Sekunde dauern.

Gleichzeitig soll der Computergegner natürlich so stark wie möglich sein. Es soll in Zukunft zwar auch verschiedene Schwierigkeitslevel geben, diese werden aber durch ein künstliches Abschwächen des optimalen Computergegners umgesetzt und spielen hier keine Rolle.

Im Rahmen dieser Arbeit liegt der Fokus auf “klassischen” algorithmischen Verfahren zur Umsetzung von Computergegnern. Verschiedene Algorithmen werden im Verlaufe der Arbeit vorgestellt, implementiert und miteinander verglichen. Am Ende wird der beste Algorithmus ermittelt und in die App eingebaut.

2 Chamäleon Schach

Entstehung

“Chamäleon Schach” ist ein schachähnliches Brettspiel, welches von Wolfgang Großkopf, dem Großvater des Autors, im Jahre 1982 entwickelt wurde. Die erste Version war damals komplett aus Holz gefertigt.



Abbildung 2.1: Der Holzkasten der ersten Version von “Chamäleon”



Abbildung 2.2: Mitten im Spiel von “Chamäleon”

Einen ersten Versuch das Spiel zu verlegen, hat es bereits zu DDR-Zeiten gegeben. Die

Qualität der gefertigten Spiele in der DDR ist aber sehr schlecht gewesen. Daher haben viele Spieleautoren kaum Anstrengungen zur Veröffentlichung unternommen und eher Spiele für sich selbst im privaten Bereich entwickelt und gebaut. (vgl. Geithner und Thiele 2013)

Erst mit der Wende ist das Spiel im VSK Verlag in ordentlicher Qualität erneut verlegt worden. Dabei hat es zwei Auflagen gegeben, eine 1990 und eine weitere 1992. 1991 belegte Chamäleon, wie das Spiel damals hieß, den siebten Platz im bekannten Ausscheid “Spiel des Jahres”. (vgl. Großkopf und Schubert 2019, s. 31)

Kurz vor dem Tod des Großvaters ist die Idee entstanden, das Brettspiel in digitaler Form neu aufzulegen. Der Enkel und Autor der Arbeit ist mit dieser Aufgabe betraut worden. Es existiert bereits eine Software-Bibliothek, welche die Spiellogik implementiert und ein erster App-Prototyp ist ebenfalls entstanden. Aus Vermarktungsgründen ist das Spiel nun umbenannt worden in “Chamäleon Schach”, um die nahe Verwandtschaft zum traditionellen Schach zu verdeutlichen.

Spielprinzip

Die ausführlichen Spielregeln finden sich im Anhang. An dieser Stelle werden lediglich die wichtigsten Kernaspekte des Spieles erläutert, welche für die algorithmische Lösung wichtig sind.

Gespielt wird auf einem speziellen Schachbrett mit 8x8 Feldern. Anders als beim klassischen Schach, haben die Felder aber vier unterschiedliche Farben (rot, grün, gelb und blau). Die Farben sind nach einem vorgegebenen Muster über das Brett verteilt.

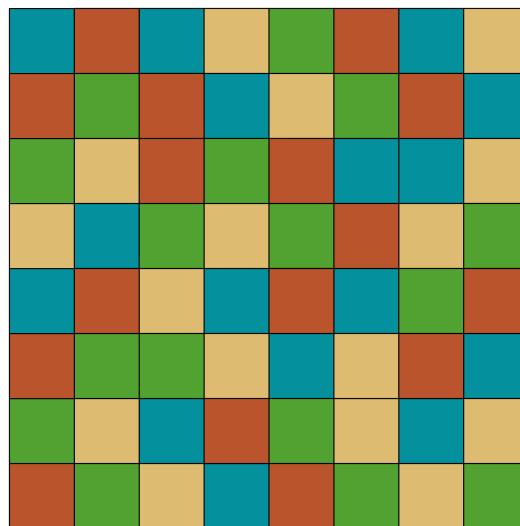


Abbildung 2.3: Das Spielbrett von “Chamäleon Schach”

Die Spielfiguren sind sozusagen Chamäleons (also gestaltwandelnde Echsen). Denn je nach der Farbe des Feldes, auf dem sich eine Figur befindet, hat die Figur eine andere Schach-Rolle. Es kommen vier verschiedene Rollen vor: Springer, Königin, Läufer und Turm. Die Zuordnung von Feldfarbe zu Rolle ist je nach Figur unterschiedlich, ändert sich aber nicht im Verlaufe des Spieles.



Abbildung 2.4: Die Spielfiguren von Spieler Rot mit ihren Farbe-Rolle-Zuordnungen

Jeder Spieler startet mit vier Figuren in einer festen Startaufstellung. Die Spieler ziehen nacheinander im Uhrzeigersinn und bewegen stets eine ihrer Figuren. Gegnerische Figuren können, wie beim richtigen Schach, geschlagen werden. Es besteht Zug-, aber kein Schlagzwang.

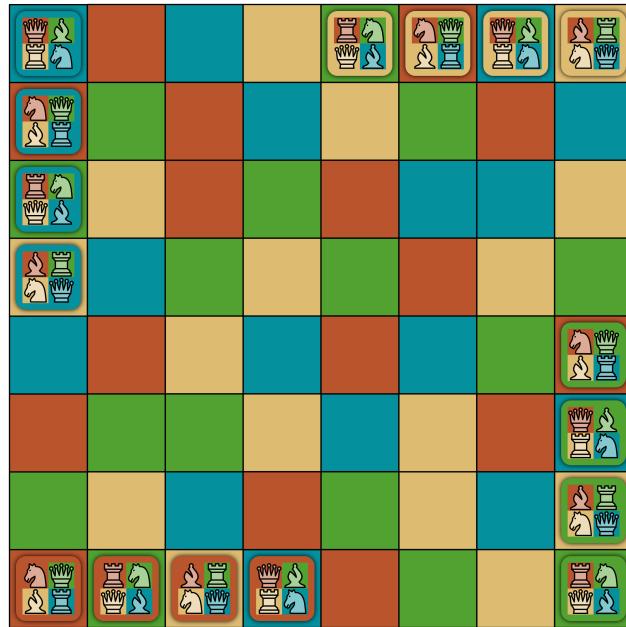


Abbildung 2.5: Die Startaufstellung der Figuren bei einem Vier-Spieler-Spiel

Im Verlauf des Spieles schrumpft das Brett, um die Spieler zusätzlich unter Druck zu setzen. Die Brettgröße wird von den äußersten Figuren bestimmt. Wird durch das Bewegen einer Figur eine oder mehrere der äußeren Reihen frei, so schrumpft das Brett entsprechend.

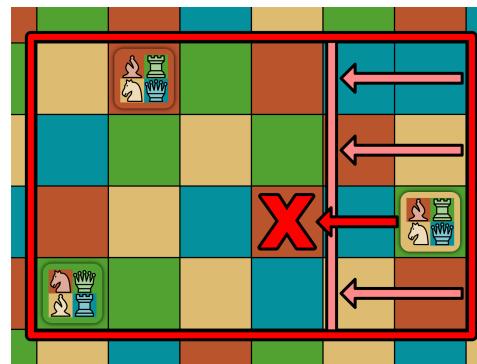


Abbildung 2.6: Das Schrumpfen des Spielbretts

Ziel des Spieles ist es, alle gegnerischen Figuren zu schlagen und damit der einzige Überlebende zu sein.

3 Spieltheorie

Um einen intelligenten Computergegner zu programmieren, muss zuerst ermittelt werden, was überhaupt eine intelligente Spielweise ist. Dazu gibt es das mathematische Feld der Spieltheorie, die zur Analyse eben dieses Problems verwendet wird.

Allgemein

Einen sehr guten Überblick und Einstieg in die Spieltheorie gibt Riechmann (2014).

Die Spieltheorie ist eine spezielle Form der Entscheidungstheorie. Der grundsätzliche Aufbau eines Entscheidungsproblems umfasst mehrere Komponenten:

- Es gibt einen Entscheider, der sich in einem bestimmten Umweltzustand (engl. state) befindet.
- Der Entscheider hat mindestens zwei verschiedene Handlungsalternativen (engl. actions), zwischen denen er vollkommen frei wählen kann.
- Mit der Entscheidung entsteht ein Handlungsergebnis (engl. result). Dieses Ergebnis wird typischerweise mit numerischen Werten beschrieben, welche eine Auszahlung (engl. utility/payout) für den Entscheider darstellt.
- Der Entscheider hat ein bestimmtes Ziel; in den meisten Fällen ist es die Maximierung seiner Auszahlung.

Die normative Entscheidungstheorie versucht nun die Entscheidungsfindung zu unterstützen. Es gilt also mit geeigneten Verfahren und Analysen herauszufinden, welche Entscheidung in welcher Situation getroffen werden sollte, um das bestmögliche Ergebnis für den Entscheider zu erzielen. (vgl. Riechmann 2014, p. 1)

Spiele – vor allem Brettspiele – funktionieren grundsätzlich genauso:

- Jeder Spieler ist ein Entscheider, welcher durch die Spielregeln verschiedenen Spielsituationen (engl. game states) ausgesetzt wird.
- Die Spieler haben in den Situationen verschiedene mögliche Züge/Handlungen, zwischen denen sie wählen können. Je nach Spiel werden die Entscheidungen gleichzeitig oder nacheinander getroffen. Häufig führt auch eine getroffene Entscheidung in einen neuen Spielzustand, in dem wiederum eine Entscheidung zu treffen ist. Erst später wird dann ein Endzustand (engl. terminate state) erreicht, mit dem das Spiel endet.
- Das Ergebnis kann aus monetären Werten, Punkten oder simplen Ergebnisbeschreibungen (gewonnen, unentschieden, verloren) bestehen. Bei einigen Spielen erfolgen bereits während des Spielverlaufes Teilauszahlungen, bei anderen Spielen gibt es erst ganz am Ende nach einer langen Kette von Entscheidungen eine finale Auszahlung.
- Ziel eines jeden Spielers ist die Maximierung seines Endergebnisses.

Genauso wie die normative Entscheidungstheorie ist es auch das Ziel der Spieltheorie herauszufinden, wie sich ein Spieler entscheiden sollte, um das bestmögliche Ergebnis zu erzielen. Allerdings sind Spiele häufig wesentlich komplexer als normale Entscheidungsprobleme. Ebenso sind verschiedenste Methoden und Verfahren nötig, um unterschiedliche Arten von Spielen lösen zu können. All dies ist ebenfalls Teil der Spieltheorie.

Die formale Beschreibung erfolgt in dieser Arbeit wie folgt:

- s bezeichnet einen konkreten Spielzustand und S die Gesamtheit aller Spielzustände (engl. state space).
- a bezeichnet eine mögliche Handlungsalternative und A die Gesamtheit der möglichen Handlungen (engl. action space).

- die Auszahlung entsteht aus dem Paar des zugrundeliegenden Zustandes und der gewählten Aktion (engl. state-action pair) und wird als Funktion mit $\vec{v}(s, a)$ beschrieben. Das Ergebnis von \vec{v} ist ein Vektor. Je eines der Elemente von \vec{v} ist einem Spieler zugeordnet und beschreibt die Auszahlung für den jeweiligen Spieler. (v ist abgeleitet vom engl. value)
- \vec{V} beschreibt entsprechend die Gesamtheit aller Auszahlungen (engl. result space).

Einordnung von “Chamäleon Schach”

Um herauszufinden, mit welchem Verfahren “Chamäleon Schach” zu analysieren ist, muss zunächst die Art des Spieles bestimmt werden:

“Chamäleon Schach” ist ein zugbasiertes Spiel. Das heißt, dass die Spieler ihre Züge nacheinander ausführen. Es gibt keine gleichzeitigen Entscheidungen. Ein Spieler ist an der Reihe, wählt aus den verschiedenen möglichen Zügen einen aus und führt diesen durch. Dadurch ändert sich der Spielzustand und der nächste Spieler ist an der Reihe. Die verfügbaren Handlungsalternativen für einen Spieler sind also direkt von den vorherigen Zügen abhängig. Man spricht von einem *sequentiellen Spiel*. (vgl. Riechmann 2014, p. 47)

Formal bedeutet ein sequentielles Spiel, dass ein Folgezustand s' direkt von dem vorherigen Zustand s und der jeweiligen ausgeführten Aktion a abhängig ist. Dadurch kann die Auszahlungsfunktion $\vec{v}(s, a)$ auch alternativ als $\vec{v}(s')$ definiert werden. Hierbei gilt stets: $\vec{v}(s, a) = \vec{v}(s')$. Beide Versionen sind gleichwertig und werden im Folgenden beliebig verwendet.

Mit einem Blick auf das Spielbrett können alle Informationen zum Spiel gewonnen werden. Es gibt keine Informationen, die nur einzelnen Spielern zugänglich sind (das wäre zum Beispiel der Fall, wenn Karten auf der Hand gehalten werden). Ebenso gibt es keine zufälligen Einflüsse (beispielsweise durch Würfel). Jeder Spieler hat zu jeder Zeit vollen Einblick auf das gesamte Spielgeschehen. Man spricht von einem *Spiel mit perfekter Information* (engl. perfect-information game). (vgl. Luce und Raiffa 1957, pp. 58 - 59)

In “Chamäleon Schach” gibt es kein Geld, keine Punkte und keine Teilauszahlungen oder Zwischenergebnisse. Am Ende des Spieles gibt es lediglich **einen** klaren Gewinner, die restlichen Spieler haben verloren. Kooperationen zwischen den Spielern machen daher keinen Sinn. Ebenso ist klar, dass der Gewinn eines Spielers automatisch der Verlust der anderen bedeutet. “Chamäleon Schach” ist damit ein *strikt kompetitives Spiel* (engl. strictly competitive game). Solche Spiele werden auch als Nullsummenspiele (engl. zero-sum games) bzw. Spiele konstanter Summe (engl. constant-sum games) bezeichnet. (vgl. Luce und Raiffa 1957, pp. 78, 164)

“Chamäleon Schach” kann zu zweit, zu dritt oder zu viert gespielt werden. Es handelt sich also um ein *Mehrspieler-Spiel* (engl. multiplayer game). (vgl. Luce und Raiffa 1957, p. 161)

Spielanalyse

Für jedes strikt kompetitive Spiel mit perfekter Information gibt es eine klar überlegene perfekte Spielweise. (vgl. Luce und Raiffa 1957, p. 82, zitiert nach; Morgenstern und Neumann 1947)

Spielbaum (engl. game tree)

Die Analyse und Lösung eines sequentiellen Spieles erfolgt über den Aufbau eines sog. *Spielbaumes* (engl. game tree). (vgl. Riechmann 2014, p. 48)

In einem Spielbaum werden alle möglichen Spielverläufe in einer Baumstruktur dargestellt. Die Knoten des Baumes stellen die Spielzustände dar. Die Kanten symbolisieren die möglichen Handlungsalternativen des aktuellen Spielers. Mit einer Handlung/Zug gelangt man direkt von einem Spielzustand in den nächsten. Der Startzustand des Spieles bildet dabei die Wurzel

des Baumes, von welchem sich alle weiteren Handlungen und Zustände ableiten. (vgl. Luce und Raiffa 1957, p. 57)

Zum besseren Verständnis wird dieses Verfahren beispielhaft nun auf das Spiel TicTacToe angewandt. TicTacToe ist ebenfalls ein strikt kompetitives, sequentielles Spiel mit perfekter Information. Auf Grund der Einfachheit des Spieles eignet es sich sehr gut, um das Verfahren besser zu verstehen und nachvollziehen zu können.

Die Regeln sind sehr simpel: Auf einem Brett mit 3x3 Feldern platzieren nacheinander zwei Spieler ihr jeweiliges Zeichen auf ein noch freies Feld. Der eine Spieler ist Kreuz (x) und der andere ist Kreis (o). Wer zuerst drei seiner Symbole in einer Reihe, einer Spalte oder einer Diagonale gesetzt hat, gewinnt. Sollte dies nach neun Zügen noch keinem der Spieler gelungen sein, dann endet das Spiel mit einem Unentschieden, da kein freies Feld mehr übrig ist. (vgl. Abu Dalfaa, Abu-Nasser, und Abu-Naser 2019)

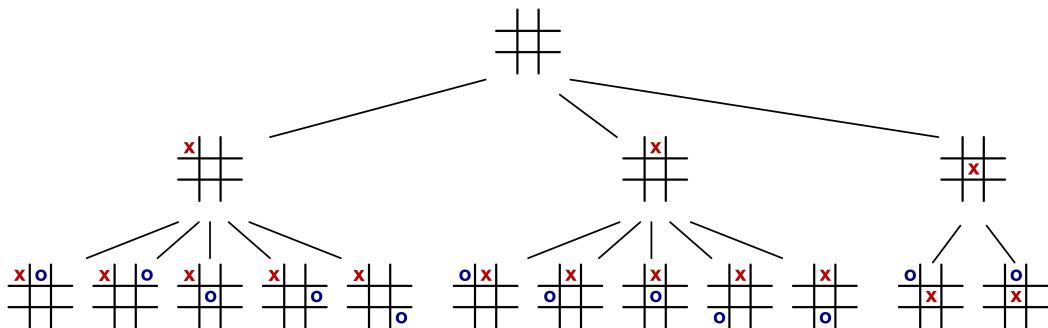


Abbildung 3.1: Spielbaum von TicTacToe für die ersten beiden Züge

Die Abbildung 3.1 zeigt den Spielbaum für TicTacToe für die ersten zwei Züge. Der Startzustand ist das leere Brett. Der erste Spieler (Kreuz) hat nun drei verschiedene Züge zur Auswahl. Theoretisch sind es neun Züge. Da das Spielbrett aber beliebig gedreht und gespiegelt werden kann, lassen sich die Möglichkeiten auf drei zusammenfassen.

Je nach Wahl von Spieler Kreuz entsteht nun eine andere Spielsituation. Die möglichen Züge von Spieler Kreis sind also von dem vorherigen Zug direkt abhängig. Dieses Prinzip wird immer so fortgesetzt, bis ein Endzustand erreicht ist, indem entweder einer der Spieler gewonnen hat oder es zu einem Unentschieden gekommen ist.

Es wird bereits hier für dieses eigentlich sehr simple Spiel deutlich, dass Spielbäume sehr schnell sehr groß werden. Das wird noch für Probleme sorgen.

Equilibrium-Punkte/Sicherheitsniveau

Nun sind alle möglichen Spielzustände, alle Aktionen und damit alle erdenklichen Spielverläufe im Spielbaum erfasst. Jetzt müssen die Auszahlungen hinzugefügt werden.

Genau wie bei "Chamäleon Schach" gibt es auch bei TicTacToe nur eine Auszahlung ganz am Ende des Spieles. Daher können vorerst nur die Endzustände mit einem Auszahlungsergebnis versehen werden. Da TicTacToe ein Zwei-Spieler-Spiel ist, enthält der Auszahlungsvektor \vec{v} zwei Elemente. Das erste Element beschreibt die Auszahlung für Spieler Kreuz, das zweite für Spieler Kreis. Im Falle des Sieges wird +1, im Falle der Niederlage -1 und für ein Unentschieden 0 ausgezahlt. (vgl. Luce und Raiffa 1957, pp. 59 - 60)

Für TicTacToe gibt es nur drei mögliche Endergebnisse:

- Kreuz hat gewonnen: $\vec{v} = (+1, -1)$
- Kreis hat gewonnen: $\vec{v} = (-1, +1)$
- unentschieden: $\vec{v} = (0, 0)$

In allen Fällen beträgt die Summe der Auszahlung aller Spieler 0. Daher wird diese Konstellation als Nullsummenspiel bezeichnet. Bzw. als Spiel mit konstanter Summe, da die Werte

für die Auszahlung natürlich beliebig gewählt werden können. Alle strikt kompetitiven Spiele verhalten sich so, da der Gewinn des einen, den Verlust des anderen Spielers bedeutet. (vgl. Luce und Raiffa 1957, p. 164)

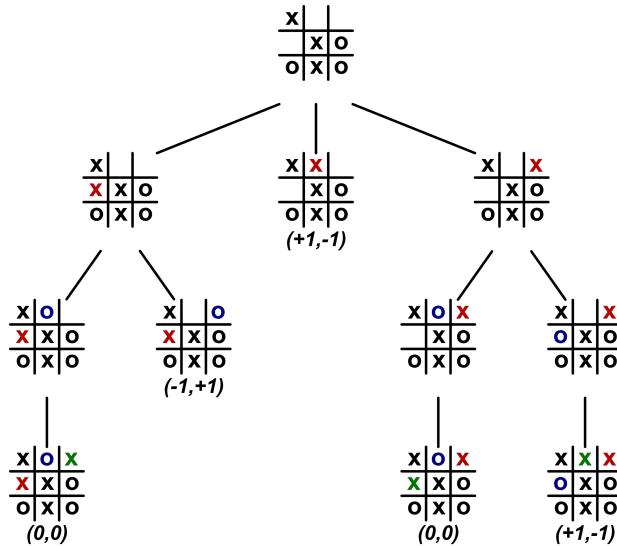


Abbildung 3.2: Ausschnitt vom Spielbaum von TicTacToe gegen Ende des Spieles mit Auszahlungen

Die Abbildung 3.2 zeigt einen Ausschnitt des Spielbaumes von TicTacToe gegen Ende des Spieles mit den Auszahlungen in den Endzuständen. Da es sich um ein Spiel mit perfekter Information handelt, ist der einzige Faktor, der das Ergebnis des Spieles beeinflusst, die Entscheidungen der Spieler. Wenn nun von "rationalen" Spielern ausgegangen wird (Spieler, deren einziges Ziel die Maximierung ihrer Auszahlung ist), kann jedem verbleibenden Spielzustand ebenfalls eine klare zu erwartende Auszahlung zugeordnet werden. Dies erfolgt über eine Rückwärtsauflösung der Auszahlungen in den Endzuständen. (vgl. Luce und Raiffa 1957, p. 82; Morgenstern und Neumann 1947)

In der Abbildung 3.2 ist zu sehen, dass drei Spielzustände auf den unteren Ebenen noch keine Auszahlung haben. Tatsächlich gibt es in diesen Zuständen aber auch keine Entscheidung zu treffen, da Spieler Kreuz hier sowieso nur ein freies verbleibendes Feld zur Verfügung hat. Das heißt, die Auszahlung aus den Endzuständen (ein Zug weiter) kann direkt den vorherigen Zuständen zugeordnet werden, da es ja keine andere Möglichkeit des Spielverlaufes mehr gibt. Dies ist in Abbildung 3.3 dargestellt.

Interessant ist die Auflösung bei Knoten, wo es mehr als eine mögliche Aktion gibt. Dazu sei der linke, noch nicht bewertete Knoten in Abbildung 3.3 betrachtet. Spieler Kreis ist am Zug. Auf dem linken Zweig würde es zu einem Unentschieden kommen (Auszahlung für Kreis: 0), auf dem rechten würde Kreis gewinnen (Auszahlung: +1). Wenn Spieler Kreis rational spielt, wird er sich natürlich für den rechten Zweig entscheiden. Daher kann dem besagten Knoten die Auszahlung (-1, +1) zugordnet werden, weil das die zu erwartende Auszahlung ist. Abbildung 3.4 zeigt die Auszahlungen für alle Knoten in diesem Ausschnitt des Spielbaums.

Aus diesen Ergebnissen lässt sich nun eine Funktion $\bar{v}^*(s)$ definieren, die also zu jedem Spielzustand die zu erwartende Auszahlung zurückgibt. Der * soll hierbei symbolisieren, dass von einer perfekten Spielweise ausgegangen wird.

Diese Auszahlungen werden auch als **Sicherheitsniveaus** bezeichnet. Es handelt sich ja um das Endergebnis, welches ein Spieler auf jeden Fall noch erreichen kann. Eventuell könnte es sogar besser ausfallen, wenn seine Gegner Fehler machen oder nicht rational spielen. (vgl. Riechmann 2014, pp. 81 - 83)

Sicherheitsniveaus werden teilweise auch als **Equilibrium-Punkte** bezeichnet. Jeder Spieler

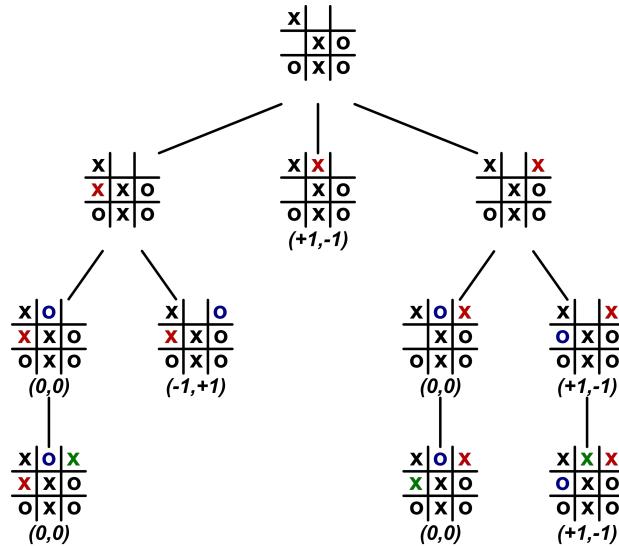


Abbildung 3.3: Durchschleifen der Auszahlungen bei Zügen ohne Entscheidung

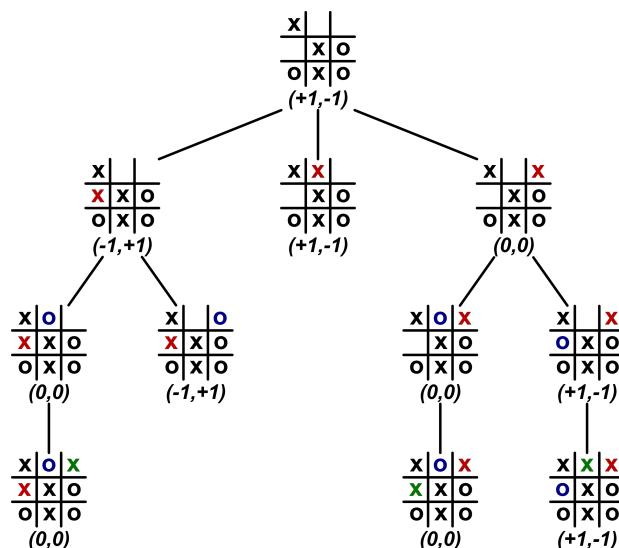


Abbildung 3.4: Auszahlungen für alle Knoten in diesem Ausschnitt des Spielbaums

maximiert stets sein Endergebnis. Da es sich um strikt kompetitive Spiele handelt, bedeutet das Maximieren des eigenen, automatisch das Minimieren der Ergebnisse der Gegner. Das Ergebnis von \vec{v}^* beschreibt nun das Gleichgewicht, welches sich durch das ständige Wechselspiel von Maximierung und Minimierung ergibt – eben einen Equilibrium- bzw. Gleichgewichts-Punkt (vgl. Luce und Raiffa 1957, p. 78)

Perfekte Spielweise

Mit den zu erwartenden Auszahlungen aus $\vec{v}^*(s)$ ist nun die Umsetzung eines perfekten Spielers denkbar einfach: In einem beliebigen Spielzustand müssen lediglich die möglichen Züge bzw. die daraus resultierenden Spielzustände betrachtet werden. Derjenige, welcher die höchste zu erwartende Auszahlung für den Spieler bringt, ist der Zug, der durchgeführt werden sollte. (vgl. Sturtevant 2003, p. 29)

Formal definiert, heißt das:

$$a^* = \operatorname{argmax}_a \vec{v}^*(s, a)$$

Wobei a^* der optimale Zug ist und s der Spielzustand, in dem sich der Spieler jetzt gerade befindet.

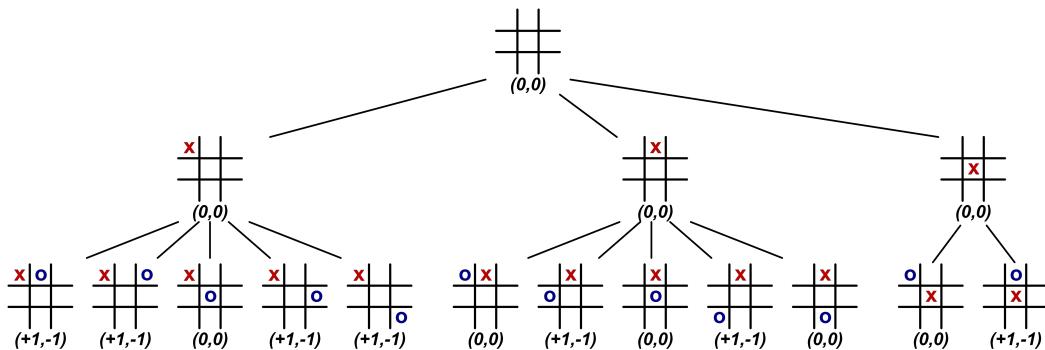


Abbildung 3.5: Spielbaum von TicTacToe mit Auszahlungen

Theoretisch reicht diese Definition aus. Praktisch stellt sich noch die Frage, wie mit gleichwertigen Auszahlungen umgegangen werden soll. Dazu ist in Abbildung 3.5 der Spielbaum von TicTacToe zu Beginn des Spieles mit Auszahlungen dargestellt. Alle möglichen Eröffnungszüge sind mit $(0,0)$ bewertet. An sich macht es also keinen Unterschied, welcher Zug gewählt wird. Auf der nächsten Ebene im Spielbaum (einen Zug weiter) fällt allerdings ein Unterschied auf: Auf dem linken Zweig hat Spieler Kreis nur einen von fünf Zügen, auf dem er ein Unentschieden erwirken kann. Menschliche Spieler machen ja durchaus Fehler in ihrer Spielweise. Der linke Zweig führt zwar theoretisch ebenso zu einem Unentschieden wie die anderen beiden, die Wahrscheinlichkeit eines menschlichen Fehlers ist hier aber höher. Daher gilt das Belegen des Eckfeldes allgemein als der beste Eröffnungszug im TicTacToe. (vgl. Sturtevant 2003, p. 29; Abu Dalfa, Abu-Nasser, und Abu-Naser 2019)

Dieses Thema wurde nur der Vollständigkeit halber erwähnt. Die wichtigere Frage ist, ob sich dieses Verfahren überhaupt praktisch umsetzen lässt oder ob die Komplexität (vor allem der Umfang des Spielbaumes) zu hoch ist.

Komplexität von Brettspielen

Anzahl verschiedener Spielverläufe

Es ist bereits deutlich geworden, dass das Aufbauen des kompletten Spielbaumes sehr aufwendig ist. Diese Bäume wachsen in ihrer Größe exponentiell an. Die vollständige Generierung

ist aber nötig, um die Funktion $\vec{v}^*(s, a)$ korrekt definieren zu können.

TicTacToe ist dabei noch ein harmloses Beispiel, da die Anzahl an möglichen Zügen im Verlaufe des Spieles immer weniger wird (da ja immer mehr Felder bereits besetzt sind). Außerdem gibt es ein klar definiertes Ende nach spätestens neun Zügen. Insgesamt gibt es für TicTacToe 31.896 verschiedene Spielverläufe, wobei schon Spielsituationen, die über Drehungen und Spiegelungen erzeugt werden können, herausgerechnet worden sind. (vgl. „Tic Tac Toe – Wikiludia“ o. J.) Das ist eine recht hohe Zahl, die für einen Computer aber ohne Probleme zu bewerkstelligen ist. Nun stellt sich allerdings die Frage, wie komplex ein Spiel wie “Chamäleon Schach” ist.

Für Schach, welches ja eng mit “Chamäleon Schach” verwandt ist, gibt es eine bekannte Abschätzungen der Komplexität von Claude Shannon (1950). Schach hat kein klares Ende nach einer bestimmten Anzahl von Zügen. Ebenso schwankt die Anzahl der möglichen Züge sehr stark während des Spielverlaufes. Die Schätzung wird nun so durchgeführt, dass die durchschnittliche Anzahl der möglichen Züge in einer Spielsituation (engl. branching factor) mit der durchschnittlichen Anzahl der Züge in einem kompletten Spiel (engl. depth) potenziert wird. Der branching factor beschreibt also, wie viele neue Zweige an einen bestehenden Knoten angehängt werden, wenn der nächste Zug betrachtet werden soll. Die depth schreibt, wieviele Ebenen tief der Spielbaum ist. Shannon ist in seiner Rechnung von durchschnittlich 30 möglichen Zügen und einer typischen Spieldauer von 80 Zügen ausgegangen. Seine Rechnung hat folgendermaßen ausgesehen:

$$b^d = 30^{80} = 30^{2 \cdot 40} \approx 10^{3 \cdot 40} \approx 10^{120}$$

Die Anzahl der Spielverläufe für Schach beträgt also 10^{120} . Das ist eine unglaublich hohe Zahl. Zum Vergleich: Die Anzahl der Atome im gesamten Universum wird auf circa 10^{79} geschätzt. („universe atom count – WolframAlpha“ o. J.) Das Generieren des kompletten Spielbaumes ist für Schach also selbst mit modernen Computern aussichtslos. (vgl. Shannon 1950)

Für “Chamäleon Schach” sind die Werte experimentell ermittelt worden. Dazu hat der ermittelte Sieger-Algorithmus mehrere Spiele gegen sich selbst gespielt. Zusätzlich ist mit verschiedenen Wahrscheinlichkeit ein Zufallzug ausgeführt worden. Dadurch sollten die Spiele abwechslungsreicher verlaufen und menschliche Fehler simuliert werden. Anschließend ist aus allen Spielen der durchschnittliche branching factor und die durchschnittliche Spieldauer ermittelt worden. Die Ergebnisse sind der Tabelle zu entnehmen:

Tabelle 3.1: Die Komplexität von “Chamäleon Schach”

Spielart	b	d	b^d
2 Spieler	16	30	10^{36}
3 Spieler	16	50	10^{60}
4 Spieler	16	56	10^{67}

Mit 10^{67} möglichen Spielverläufen ist die Komplexität von “Chamäleon Schach” ebenfalls astronomisch hoch. Selbst modernste Rechner würden für diese Berechnung sicher einige Jahre brauchen. Auf mobilen Endgeräten ist dieser Ansatz überhaupt nicht umsetzbar.

Hinweis: Die Werte sind selbstverständlich erst im Nachhinein ermittelt worden. Von Anfang an war aber klar, dass die Komplexität von “Chamäleon Schach” sehr hoch sein wird. Die genauen Werte, auf denen diese Berechnung beruht, finden sich im angehängten Git-Repository in einem Jupyter-Notebook-Dokument.

Anzahl verschiedener Spielsituationen

Ein alternativer Ansatz könnte darin bestehen, die Berechnung nur einmal auf sehr leistungsfähigen Computern durchzuführen und die Ergebnisse z.B. in einer Hash-Table zu speichern.

Dazu muss geklärt werden, wieviele verschiedene Spielbrettzustände es geben kann, da dies den benötigten Speicherplatz bestimmt.

TicTacToe ist mit 765 verschiedenen Anordnungen wieder sehr überschaubar. („A008907 – OEIS“ o. J.)

Für Schach gibt es eine grobe Hochrechnung der möglichen Spielbrettanordnungen, ebenfalls von Claude Shannon (1950). Er beziffert Schach mit 10^{43} verschiedenen Anordnungen – wieder eine astronomisch hohe Zahl.

Für „Chamäleon Schach“ kann die Anzahl der möglichen Anordnungen ebenfalls gut abgeschätzt werden. Bis zu vier Spieler können mitspielen. Jeder Spieler hat vier klar voneinander unterscheidbare Figuren. Da die Figuren ja geschlagen werden können, muss ermittelt werden, wieviele Möglichkeiten es gibt, ein bis 16 Figuren auf einem 8x8 Schachbrett anzurufen. Hier die Formel dazu:

$$\sum_{i=1}^{16} \left(\binom{64}{i} \cdot \frac{16!}{(16-i)!} \right) \approx 10^{28}$$

Es ist also deutlich zu sehen, dass der Ansatz aus der Spieltheorie für „Chamäleon Schach“ weder berechenbar, noch die Ergebnisse der Berechnung speicherbar sind. Daher muss für die Umsetzung ein anderes Verfahren verwendet werden. In dieser Arbeit wird dazu der sog. heuristische Ansatz verwendet. Damit beschäftigt sich nun der restliche Teil.

4 Heuristischer Ansatz

Das Wort Heuristik beschreibt eine “methodische Anleitung [...] zur Gewinnung neuer Erkenntnisse” (Dudenredaktion o. J.). Es stammt von dem griechischen *heuriskein* ab, was “finden” oder “entdecken” bedeutet. Heuristische Methoden versuchen also aus den vorhandenen (häufig unvollständigen) Informationen neue Erkenntnisse zu gewinnen.

Prinzip

Heuristische Bewertungsfunktion

Der heuristische Ansatz nutzt im Grundsatz das gleiche Vorgehen wie die Spieltheorie. Der Unterschied besteht in der Definition der Funktion $\vec{v}^*(s, a)$. Anstatt die zu erwartende Auszahlung über die Generierung und Rückwärtsauflösung des kompletten Spielbaums zu ermitteln, erfolgt die Bewertung anhand des aktuellen Spielzustandes. Es wird versucht, mittels bestimmter Metriken der aktuellen Spielsituation eine Vorhersage über das Endergebnis zu treffen. (vgl. Sturtevant 2003, pp. 26 - 27)

Die Metriken, die zur Schätzung des Endergebnisses verwendet werden, sind vom jeweiligen Spiel abhängig. Sie werden von Menschen anhand von Spielerfahrung und einer umfassenden Analyse der Spielregeln erstellt. Häufig ist auch der Vergleich verschiedener Metriken nötig, um eine adäquate Bewertungsfunktion zu finden. (vgl. Sturtevant 2003, p. 27)

Beispiele für solche Metriken könnten die Anzahl der Spielfiguren eines Spielers sein oder wie weit ein Spieler schon auf dem Brett vorangekommen ist oder wieviele Stiche er im Vergleich zu den anderen gemacht hat etc.

Diese **heuristische Bewertungsfunktion**, die im Folgenden als $\vec{h}(s, a)$ bezeichnet wird, ist der erste Baustein des heuristischen Ansatzes.

Beschränktes Durchsuchen des Spielbaums

In der Spieltheorie wird die Funktion $\vec{v}^*(s, a)$ verwendet, um direkt die möglichen Züge mit einer Auszahlung zu versehen. Dann wird der Zug mit der besten Auszahlung ausgewählt. Die Ergebnisse der Funktion kommen dadurch zustande, dass der Spielbaum bis zum Ende aufgebaut und die Auszahlungen rückwärts aufgelöst werden. Daher ist es auch naheliegend, dass die Effektivität der Bewertungsfunktion im heuristischen Ansatz gesteigert werden kann, wenn der Spielbaum wenigstens teilweise über ein paar Züge aufgebaut wird.

Im heuristischen Ansatz ist das Vorgehen nun so, dass von der aktuellen Spielsituation aus der Spielbaum über eine festgelegte Tiefe aufgebaut wird. Die heuristische Bewertungsfunktion wird verwendet, um den untersten Knoten eine zu erwartende Auszahlung (Schätzung) zuzuordnen. Anschließend werden die Auszahlungen, wie in der Spieltheorie, rückwärts bis zur ersten Ebene aufgelöst. Dadurch erhalten die Knoten auf der ersten Ebene (welche ja die möglichen Züge sind) eine bessere, weiter “vorausschauende” Auszahlung zugeordnet. Anschließend wird ebenfalls der Zug mit der besten Auszahlung für den Spieler ausgeführt. Dieses Vorgehen verbessert die eher wagen Schätzungen der Bewertungsfunktion immens. (vgl. Sturtevant 2003, p. 41)

Pruning

Spielbäume werden in der Regeln mittels einer Tiefensuche aufgebaut. Manchmal ist während der Generierung eines Zweiges schon absehbar, dass dieser keine besseren Auszahlungen liefern

wird, als die bereits durchsuchten vorherigen Zweige. Wenn das auftritt, kann die restliche Generierung des aktuellen Zweiges auch abgebrochen und mit dem nächsten fortgesetzt werden. Dieser Vorgang wird als **Pruning** (engl. beschneiden) bezeichnet. Durch das Pruning kann Rechenzeit gespart werden. (vgl. Sturtevant 2003, p. 41)

Wie genau das Pruning eingesetzt werden kann, hängt von dem konkret verwendeten Algorithmus ab.

Alle Algorithmen haben ohne Pruning eine Laufzeit von $\mathcal{O}(b^d)$. Wobei b der branching factor, also die Anzahl der möglichen Züge, die ein Spieler ausführen kann, ist. d ist die Tiefe (engl. depth), also wieviele Züge in die Zukunft geschaut wird. Mit Pruning kann sich diese Laufzeit verbessern. Der Worst Case bleibt aber stets bei $\mathcal{O}(b^d)$.

Iterative Deepening

Die Komplexität des Spielbaums wächst mit jeder weiteren Ebene exponentiell an. Entsprechend steigt auch die benötigte Rechenzeit exponentiell. Ein typisches Vorgehen besteht deshalb darin, dass der Spielbaum erstmal nur für eine Ebene generiert und bewertet wird. Steht dann noch Rechenzeit zur Verfügung, wird eine weitere Ebene hinzugefügt und die bisherigen Auszahlungen werden entsprechend aktualisiert. Steht dann immer noch Rechenzeit zur Verfügung, kommt eine weitere Ebene hinzu usw. Dieser Vorgang wird als **Iterative Deepening** bezeichnet. (vgl. Sturtevant 2003, pp. 91 - 92)

Durch diesen Vorgang wird die Genauigkeit der Berechnung der erwarteten Auszahlungen immer besser, je mehr Zeit zur Verfügung steht. Damit lässt sich z.B. auch die Stärke eines Computergegners steuern, indem er einfach mehr oder weniger Rechenzeit zur Verfügung hat. Durch effektivere Pruning-Verfahren können die Algorithmen in der gleichen Zeit eine höhere Suchtiefe des Spielbaums erreichen und werden dadurch stärker.

Algorithmen

Für die konkrete Umsetzung der Konzepte des heuristischen Ansatzes gibt es eine Vielzahl an konkreten Algorithmen. Alle unterscheiden sich ein wenig in ihrer Arbeitsweise, der Möglichkeiten des Pruning, Eignung für verschiedene Arten von Spielen usw. Tatsächlich gibt es für strikt kompetitive Mehrspieler-Spiele mit perfekter Informationen keinen einzelnen Algorithmus, der pauschal für alle diese Spiele geeignet ist. Stattdessen müssen zumeist mehrere dieser Algorithmen implementiert und gegeneinander getestet werden. (vgl. Sturtevant 2003, pp. 3 - 4)

Im Folgenden werden nun die Algorithmen vorgestellt, die auch für "Chamäleon Schach" implementiert und getestet worden sind. In diesem Kapitel werden die Algorithmen und ihre Arbeitsweise allgemein beschrieben. Die konkrete Umsetzung für "Chamäleon Schach" folgt dann im nächsten Kapitel

Minimax mit Alpha-Beta Pruning

Prinzip

Der **Minimax-Algorithmus mit Alpha-Beta Pruning** ist die Standardlösung für strikt kompetitive Spiele mit perfekter Informationen – solange es sich um Zwei-Spieler-Spiele handelt. (vgl. Sturtevant 2003, pp. 3 - 4)

Der Vorteil bei dieser Variante liegt darin, dass die Auszahlung als Skalar dargestellt werden kann (vgl. Luckhart und Irani 1986). Ein Spieler ist Max, der andere ist Min. Die Auszahlung wird nun so definiert, dass positive Auszahlungen für Spieler Max von Vorteil sind und negative entsprechend für Spieler Min. Bei der Rückwärtsauflösung der Auszahlungen wählt Spieler Max also stets die höchstmögliche Auszahlung (er maximiert das Ergebnis) und Spieler Min stets die kleinstmögliche Auszahlung (er minimiert das Ergebnis). Dies ist zulässig, da es sich ja um ein strikt kompetitives Spiel handelt. (vgl. Sturtevant 2003, pp. 27 - 28)

Alpha-Beta Pruning

Hinzu kommt die Anwendung eines sehr effektiven Pruning-Verfahrens: dem **Alpha-Beta Pruning**.

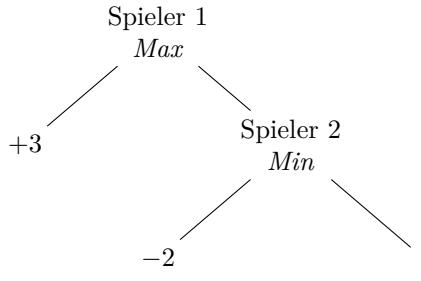


Abbildung 4.1: Alpha-Beta Pruning

Die Abbildung 4.1 zeigt einen einfachen Spielbaum mit skalaren Auszahlungen. Spieler 1 ist Max und hat auf dem zuerst betrachteten Zweig (links) eine +3 als Bewertung erhalten. Spieler 1 würde sich also für diesen Zweig entscheiden, wenn keine höhere Bewertung gefunden wird. Auf dem nächsten Zweig (ein Zug weiter in der Zukunft) würde Spieler 2, welcher Min ist, eine -2 als Auszahlung erhalten. Da Spieler 2 immer die kleinstmögliche Auszahlung wählt, wird er auf jeden Fall den Zweig mit der -2 oder sogar eine noch kleinere Bewertung wählen. Spieler 1 steht aber eine Ebene darüber und kann auf dem linken Zweig bereits ein +3 bekommen. Damit ist klar, dass Spieler 1 niemals den rechten Zweig wählen wird, sondern stets den linken. Der komplette rechte Zweig kann also ignoriert werden, sobald die -2 gefunden wurde. Das ist das Alpha-Beta Pruning. (vgl. Sturtevant 2003, pp. 41 - 42)

Der Name leitet sich davon ab, dass in einer Variable namens α der beste bisher gefundene Wert für Spieler Max und in β der bisher beste Wert für Spieler Min gespeichert wird. α wird mit $-\infty$ initialisiert und nun von Spieler Max maximiert. Bei β erfolgt die Initialisierung entsprechend mit $+\infty$ und die Variable wird nun minimiert. In dem Moment, wo $\alpha \geq \beta$ ist, kann beschnitten werden. Denn Spieler Max würde α ja weiter maximieren, Spieler Min hat aber auf einem anderen Zweig schon eine niedrigere (für ihn bessere) Bewertung gefunden und umgekehrt. (vgl. Knuth und Moore 1975)

Durch dieses Verfahren wird eine Laufzeit von $\mathcal{O}(b^{\frac{d}{2}})$ im Best Case und $\mathcal{O}(b^{\frac{3d}{4}})$ im Average Case erreicht. (vgl. Sturtevant 2003, s. 43; zitiert nach Pearl 1984)

Paranoider Ansatz für N-Spieler

Leider reicht ab drei Spielern eine skalare Bewertung der Spielzustände nicht mehr aus. Damit funktioniert dieser Algorithmus für Spiele ab drei Spielern nicht mehr. Sturtevant und Korf (2000) geben allerdings eine Lösung dazu: nämlich den **paranoiden Ansatz**.

Der Spieler in der Wurzel des Spielbaumes (der also tatsächlich einen Zug durchführen muss) ist Spieler Max. Alle seine Gegner bilden eine Koalition gegen ihn und sind Spieler Min bzw. Team Min. Nun kann wieder der Minimax-Algorithmus mit Alpha-Beta Pruning verwendet werden. Je nachdem, ob es sich bei der Rückwärtsauflösung dann um einen Knoten von Spieler Max oder um einen Knoten eines der Spieler aus Team Min handelt, wird entweder maximiert oder minimiert.

Die Bewertungsfunktion liefert dabei zunächst einen Auszahlungsvektor mit je einem Element je Spieler. Anschließend wird die Auszahlung von Spieler Max genommen und nacheinander werden die Auszahlungen der Spieler aus Team Min abgezogen. Dadurch wird aus dem Auszahlungsvektor ein skalarer Wert. Formal bedeutet das:

$$v' = v_{max} - sum(\vec{v}_{min})$$

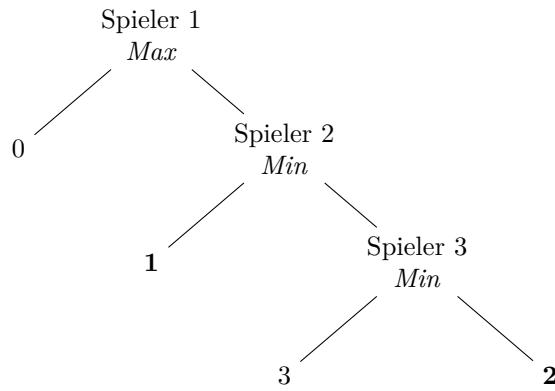


Abbildung 4.2: Rückwärtsauflösung im paranoiden Ansatz

Die Rückwärtsauflösung erfolgt wie in Abbildung 4.2 dargestellt. Der Spieler, welcher in einem Knoten am Zug ist, ist entweder Spieler Max oder gehört zum Team Min. Entsprechend wird also entweder die höchste oder die niedrigste Auszahlung gewählt. Wie in der Abbildung zu sehen ist, wird dadurch durchaus auf mehreren Ebenen des Spielbaums hintereinander nur minimiert. Es sind ja mehrere Spieler im Team Min, aber nur einer ist Max.

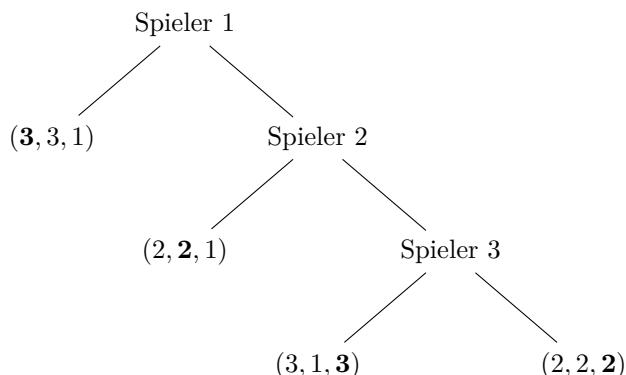
Sturtevant und Korf geben allerdings zu bedenken, dass diese Annahme zu fehlerhaftem Spielverhalten führen kann, da ja in der Realität gar keine Koalition zwischen den gegnerischen Spielern besteht. Dieses fehlerhafte Verhalten fällt außerdem mit steigender Suchtiefe immer stärker ins Gewicht. (vgl. Sturtevant 2003, pp. 132 - 133)

In Sturtevants Versuchen u.a. mit dem Spiel Chinese Checkers (2003 p. 118) ist aber der paranoide Ansatz den anderen Algorithmen häufig überlegen gewesen. Unter anderem, weil durch das effektivere Pruning eine wesentlich höhere Suchtiefe erreicht und die Bewertung der Züge dadurch "weitsichtiger" geworden ist.

Max^N

Prinzip

Der **Max^N-Algorithmus** ist von Luckhardt und Irani (1986) vorgestellt worden. Die Auszahlungen werden als Vektoren mit je einem Element je Spieler von einer heuristischen Bewertungsfunktion erzeugt. Ein beschränkter Spielbaum wird aufgebaut und die Rückwärtsauflösung erfolgt genauso, wie in der Spieltheorie beschrieben. Es wird also stets der Zweig mit der höchsten Bewertung für den Spieler ausgewählt, welcher im jeweiligen Spielzustand am Zug ist. Der Max^N ist damit der "reinste" Algorithmus, der am exaktesten das Vorgehen aus der Spieltheorie umsetzt.

Abbildung 4.3: Arbeitsweise des Max^N

Die Abbildung 4.3 zeigt die Arbeitsweise des Max^N für ein Spiel mit drei Spielern. Die Auszahlung erfolgt als Vektor mit je einem Element je Spieler. Bei der Rückwärtsauflösung werden nur die Elemente im Vektor betrachtet, welche zu dem jeweiligen Spieler gehören. Es wird stets die höchste Auszahlung für den Spieler ausgewählt. Daher stammt auch der Name Max^N . Alle Spieler maximieren ihre Auszahlung und es sind N Spieler. (vgl. Luckhart und Irani 1986)

Luckhardt und Irani (1986) haben in ihrer Arbeit aber auch gezeigt, dass ein so effektives Pruning-Verfahren, wie das Alpha-Beta Pruning, ab drei Spielern nicht mehr möglich ist. Das liegt daran, dass nicht sicher ist, ob die Elemente im Vektor in irgendeinem Bezug zueinander stehen. Wenn es sich bspw. um ein strikt kompetitives Spiel handelt und die Auszahlungsvektoren sich auch entsprechend verhalten (sie weisen eine konstante Summe auf), dann sind spezielle Pruning-Verfahren wieder möglich. Ähnlich ist es, wenn z.B. die Werte im Vektor nur steigen und nicht mehr sinken können etc. Die verschiedenen Pruning-Verfahren sind also an sehr spezielle Bedingungen geknüpft. Egal welches Verfahren dabei zum Einsatz kommen kann, keines erreicht die Effektivität des Alpha-Beta Prunings. In der Doktorarbeit von Nathan Sturtevant (2003) sind sehr viele solcher Verfahren aufgeführt. In dieser Arbeit werden allerdings nur solche vorgestellt, die tatsächlich auf "Chamäleon Schach" anwendbar und implementiert worden sind.

Immediate Pruning

Diese Art des Prunings kann angewandt werden, wenn es eine *maximale Bewertung* gibt. Eine Bewertung also, die nicht mehr übertroffen werden kann. Sobald ein Blatt im Spielbaum mit der Maximalbewertung gefunden worden ist, können alle weiteren Zweige ignoriert werden, weil sich der Wert nicht mehr verbessern kann. (vgl. Sturtevant und Korf 2000)

Im absolut besten Fall beträgt die Laufzeit dann $\mathcal{O}(b^{\frac{n-1}{n}})$. In den meisten Spielen, in denen es so einen Höchstwert gibt, kommt dieser aber eher gegen Ende vor. Zu Beginn des Spieles nützt dieses Verfahren also faktisch gar nichts.

Shallow Pruning

Die Voraussetzung für das Shallow Pruning ist, dass es eine maximale Summe der Elemente im Auszahlungsvektor gibt. Wird nun in einem Kindknoten eine Auszahlung gefunden, die zusammen mit der besten bisherigen Auszahlung im Vaterknoten die maximale Summe übersteigt, dann ist die Pruning-Bedingung erreicht. (vgl. Sturtevant und Korf 2000)

Formal definiert, lautet die Pruning-Bedingung: $v_{\text{Kind}} \geq \max\Sigma - v_{\text{Vater}}$

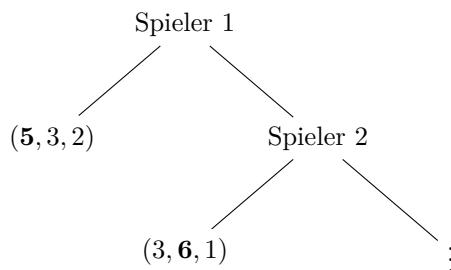


Abbildung 4.4: Shallow Pruning. $\max\Sigma = 10$

In Abbildung 4.4 ist ein Spielbaum für ein Drei-Spieler-Spiel zu sehen. Die maximale Summe der Werte im Auszahlungsvektor beträgt 10. Spieler 1 würde auf dem linken Zweig eine Auszahlung von 5 erhalten, da das erste Element im Vektor diesem Spieler zugeordnet ist. Das heißt, Spieler 1 würde sich für diesen Zweig entscheiden, wenn auf dem rechten Zweig keine bessere Auszahlung gefunden wird. Auf dem rechten Zweig hat Spieler 2 nun die Bewertung (3, 6, 1) erhalten. Damit beträgt die Auszahlung für Spieler 2 bereits 6 von insgesamt 10 möglichen Punkten. Spieler 1 kann auf dem rechten Zweig also maximal noch eine Auszahlung

von 4 erreichen. Somit ist klar, dass sich Spieler 1 niemals für den rechten Zweig entscheiden wird. Auf dem linken ist ja bereits eine viel höhere Auszahlung gefunden worden.

Diese Art des Prunings kann nur zwischen unmittelbaren Vater- und Kindknoten durchgeführt werden. Über mehrere Ebenen hinweg (sog. Deep Pruning) liefert dieses Verfahren verfälschte Ergebnisse. (vgl. Sturtevant und Korf 2000)

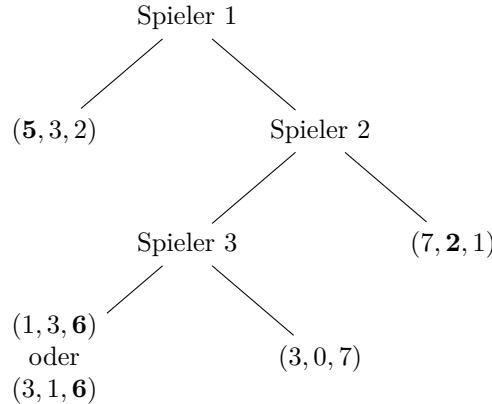


Abbildung 4.5: Deep Pruning liefert falsche Ergebnisse. $\max \Sigma = 10$

Abbildung 4.5 zeigt einen solchen Fall: Spieler 1 hat bereits eine Auszahlung von 5 auf dem linken Zweig erreicht. Nun wird mittels Tiefensuche von Spieler 3 eine Auszahlung von 6 gefunden. Es könnte angenommen werden, dass die Pruning-Bedingung erreicht ist und die restlichen Zweige nicht mehr betrachtet werden müssen. Auf dem linken Zweig von Spieler 3 sind nun zwei verschiedene Auszahlungsvektoren dargestellt. Beide würden die Pruning-Bedingung erfüllen, aber je nachdem welcher Vektor auftritt, ist das Ergebnis der Rückwärtsauflösung verschieden:

- Würde die erste (die obere) Auszahlung gefunden werden, so würde sich Spieler 2 für den linken Zweig entscheiden. Damit bekommt *Spieler 1* auf dem rechten Zweig die Auszahlung (1, 3, 6) und würde den *linken Zweig* wählen.
- Für die untere Version der Auszahlung ist es anders: Spieler 2 würde nun den rechten Zweig wählen, da er hier eine bessere Bewertung erhält. Damit bekommt Spieler 1 auf dem rechten Zweig eine Auszahlung von (7, 2, 1). Das ist wesentlich besser als die bisherige. In diesem Fall würde *Spieler 1* also den *rechten Zweig* wählen.

Es kommen also zwei vollkommen unterschiedliche Ergebnisse heraus. Würde der Spielbaum ohne Pruning nach dem Max^N-Verfahren aufgelöst werden, so würde sich Spieler 3 in jedem der beiden Fälle für den rechten Zweig entscheiden. Bei der weiteren Auflösung würden sich Spieler 2 und schließlich Spieler 1 ebenfalls für die jeweils rechten Zweige entscheiden. Das Deep-Pruning würde also je nach Fall entweder ein richtiges oder ein falsches Ergebnis liefern.

Dadurch beschränkt sich das Pruning nur auf unmittelbar verwandte Knoten und die Effektivität sinkt drastisch. Vor allem wenn die Bewertungen recht gleichverteilt sind zwischen den Spielern, würde die Pruning-Bedingung fast nie erfüllt werden. Im Best Case tendiert diese Lösung gegen $\mathcal{O}(b^{\frac{d}{2}})$. (vgl. Sturtevant 2003 p. 52 - 53)

Immediate und Shallow Pruning-Bedingung erzeugen

Immediate und Shallow Pruning sind an zwei Bedingungen geknüpft: es muss einen *Maximalwert* in der Bewertung geben und eine *maximale Summe*. Praktischerweise können diese Voraussetzungen in fast jeder Implementierung des Max^N-Algorithmus geschaffen werden. Die Idee dazu hat bereits Sturtevant (2003 s. 73) gehabt. Dazu ist folgende Modifikation notwendig:

Jedes einzelne Element im Auszahlungsvektor wird durch die Summe aller Elemente im Vektor geteilt. Formal bedeutet das, dass ein Auszahlungsvektor $\vec{v} \in \mathbb{R}_0^+ N$ (N ist die Anzahl der Spieler) in einen Vektor \vec{v}' durch folgende Operation überführt wird:

$$v'_i = \frac{v_i}{\sum v}$$

Durch diese Operation sind alle Bewertungen im Wertebereich von 0 bis 1 und die Summe aller Bewertungen ist ebenfalls stets 1. Der Ergebnisvektor dieser Operation stellt also die *Gewinnwahrscheinlichkeiten* der einzelnen Spieler dar. Damit sind die Bedingungen für Immediate und Shallow Pruning erfüllt. Im weiteren Verlauf wird diese Operation als **Normierung** bezeichnet werden.

Anmerkung: \vec{v} darf kein Nullvektor sein. Ebenso sind negative Werte im Vektor problematisch. Die meisten Bewertungsfunktionen liefern aber ohnehin nur positive Bewertungen. (vgl. Sturtevant 2003 p. 73)

Hypermax

Der Hypermax-Algorithmus ist von Mikael Fridenfalk (2014) vorgestellt worden. Das Ziel ist es gewesen, eine Variante des Alpha-Beta Prunings für N-Spieler-Spiele zu entwickeln.

Der entscheidende Punkt für das Alpha-Beta Pruning ist die Eigenschaft des Nullsummenspiels. Eine positive Bewertung ist für Spieler Max im gleichen Maße gut, wie sie für Spieler Min schlecht ist. Formal ist die Pruning-Bedingung, wie bereits beschrieben, als $\alpha \geq \beta$ definiert. Umgeformt könnte die Pruning-Bedingung auch so geschrieben werden: $a_1 + a_2 \geq 0$, wobei $a_1 = \alpha$ und $a_2 = -\beta$ ist. Mit anderen Worten: sobald die Summe der besten bisher gefundenen Werte der Spieler größer oder gleich der konstanten Summe des Spieles ist (in einem Nullsummenspiel ist die Summe also 0), kann der aktuell betrachtete Zweig im Spielbaum keine Verbesserung mehr bringen. Da es sich nun mal um ein Spiel konstanter Summe handelt, ist klar, dass das Übersteigen dieser Summe nicht zulässig ist.

Fridenfalk führt nun fort, dass Alpha-Beta Pruning möglich wäre, wenn es sich auch beim N-Spieler-Spiel um ein Nullsummenspiel handeln würde. Dazu gibt er eine Methode, dies zu erreichen. \vec{v} ist wieder das ursprüngliche Ergebnis der Bewertungsfunktion und \vec{v}' die modifizierte Version. Die Umwandlung in ein Nullsummenspiel (er bezeichnet dies als eine Transformation in einen Zero-Space Vektor) erfolgt, indem von allen Elementen in \vec{v} der Durchschnitt aller Elemente abgezogen wird. Also:

$$v'_i = v_i - avg(\vec{v})$$

Der Algorithmus funktioniert nun so, dass in einem Vektor (den Fridenfalk $\vec{\alpha}$ nennt) die jeweils bisher besten Ergebnisse für die jeweiligen Spieler gespeichert werden. Zu Beginn wird jedes Element in $\vec{\alpha}$ mit $-\infty$ initialisiert. Wann immer eine bessere Bewertung gefunden wird, wird das zum jeweiligen Spieler gehörenden Element in $\vec{\alpha}$ aktualisiert (maximiert). Sobald nun die Summe von $\vec{\alpha}$ größer oder gleich 0 ist, ist die Pruning-Bedingung erreicht und der aktuelle Zweig wird beschnitten.

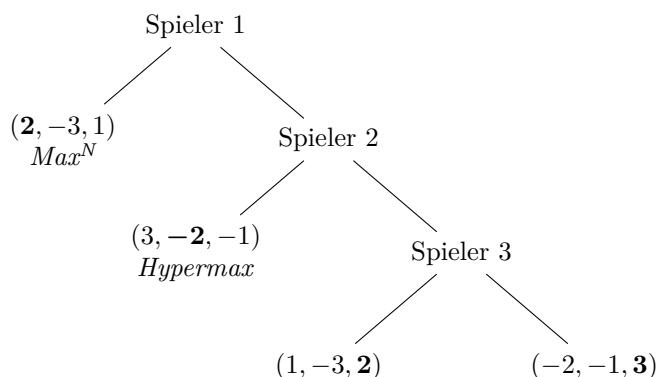


Abbildung 4.6: Unterschiede zwischen Max^N und Hypermax

In den meisten Fällen sind die Ergebnisse des Max^N und des Hypermax identisch. In einigen Fällen beschneidet der Hypermax allerdings Zweige, die Einfluss auf das Endergebnis gehabt hätten. So ein Beispiel ist in Abbildung 4.6 zu sehen. In dem Moment, wo Spieler 3 den linken Zweig absucht, ist die Pruning-Bedingung des Hypermax erreicht. Spieler 1 hat bisher eine 2 und Spieler 2 bisher eine -2 gefunden. Mit der Auszahlung von 2 für Spieler 3 ist nun die Summe der besten gefundenen Auszahlungen größer als 0. Dadurch würde sich Spieler 2 für den linken und Spieler 1 für den rechten Zweig entscheiden. Der Max^N ohne Pruning hätte sich aber korrekterweise für den rechten Zweig unter Spieler 3 entschieden (der ja vom Hypermax beschnitten worden wäre). Damit hätte auch Spieler 2 den rechten Zweig gewählt und Spieler 1 dann schließlich den linken. Also zwei vollkommen unterschiedliche Ergebnisse, je nachdem ob Pruning verwendet wird oder nicht.

Fridenfalk ist sich dieser Abweichung im Verhalten bewusst. Er argumentiert allerdings, dass in den meisten Fällen die Ergebnisse gleich sind. Außerdem kann der Hypermax durch das Pruning in der gleichen Zeit den Spielbaum tiefer durchsuchen und findet so bessere "weitsichtigere" Auszahlungen als sein Bruder-Algorithmus. Die Argumentation ist also ähnlich zum paranoiden Ansatz. Nicht die Genauigkeit der Berechnung, sondern die höhere Suchtiefe sollen den entscheidenden Vorteil liefern.

Die Unmöglichkeit von tiefen Pruning-Verfahren für N-Spieler-Spiele ist bereits von Luckhardt und Irani (1986) postuliert worden. Die Abweichungen des Hypermax beruhen u.U. auf dem gleichen Grund, warum auch das Shallow Pruning eben nur zwischen Vater- und Kindknoten korrekt funktioniert. Dies genauer zu analysieren, würde allerdings den Rahmen dieser Arbeit sprengen.

5 Implementierung

Für die Umsetzung muss zuerst eine geeignete heuristische Bewertungsfunktion ermittelt werden. Danach werden die verschiedenen vorgestellten Algorithmen implementiert. Anschließend spielen die Algorithmen gegeneinander, um den finalen Sieger-Algorithmus zu finden.

Bewertungsfunktion

Für die heuristische Bewertungsfunktion $\vec{h}(s)$ ist entscheidend, welche Faktoren in einem Spiel zu einem Sieg führen und welche Informationen während des Spielverlaufes vorliegen. (vgl. Sturtevant 2003, pp. 26 - 27)

Die Gewinnbedingung für “Chamäleon Schach” ist sehr klar: der letzte Spieler, der noch Figuren auf dem Brett hat, gewinnt. Das bedeutet: je mehr Figuren ein Spieler noch hat, desto besser stehen seine Gewinnchancen. Die Anzahl der Spielfiguren eines Spielers ist außerdem zu jeder Zeit direkt vom Spielbrett ablesbar. Die Anzahl der Figuren wird damit die Basismetrik für die Bewertungsfunktion sein.

Die Besonderheit von “Chamäleon Schach” liegt aber auch darin begründet, dass die gleiche Figur auf Feldern mit unterschiedlichen Farben auch unterschiedliche Rollen annimmt. Eine Dame kann viel weiter ziehen als ein Springer. Damit hat eine Dame auch mehr Möglichkeiten sich zu bewegen, sich in Sicherheit zu bringen, gegnerische Figuren zu schlagen usw. Daher wird auch die Rolle, welche die jeweiligen Figuren zur Zeit innehaben, wichtig für die Bewertung sein. Auch diese Information kann direkt dem Spielbrett entnommen werden.

Nun stellt sich die Frage, wie die verschiedenen Rollen zu gewichten sind. Dazu ist die folgende Tabelle erstellt worden:

Tabelle 5.1: Wertigkeiten der Schach-Rollen anhand ihrer Bewegungsweite

Rolle	Eckfeld	Mitte	Durchschnitt	Punkte
Springer	2	8	5	1
Läufer	7	14	10.5	2
Turm	14	14	14	3
Dame	21	28	24.5	5

Die möglichen Züge einer Figur sind auch abhängig davon, wo die Figur auf dem Schachbrett steht. Auf einem Eckfeld haben die meisten Figuren wesentlich weniger Zugmöglichkeiten als auf einem der vier mittleren Felder des Brettes. Die Tabelle zeigt daher die möglichen Züge einer Figur, wenn sie sich auf einem Eckfeld und in der Mitte befindet. Aus diesen Werten wird der Durchschnitt gebildet. Daraus sind die finalen Punkte abgeleitet worden, die grob die Verhältnisse zwischen den Zugmöglichkeiten der Figuren und damit ihre relative Stärke zueinander widerspiegeln.

Ein alternativer Ansatz für die Gewichtung der Rollen könnte darin bestehen, zu zählen, wieviel Züge die jeweilige Figur aktuell ausführen kann. Auf diese Art und Weise wird eine stärkere Rolle automatisch besser bewertet. Außerdem wird dadurch auch berücksichtigt, dass eine starke Rolle in einer “engen” Spielsituation gar nicht so stark ist, da ihre Bewegung eingeschränkt wird. Ein Nachteil könnte diese Methode aber auch mit sich bringen: Da die Bewertung umso besser ausfällt, je weiter sich eine Figur bewegen kann, vermeidet der Computer dadurch u.U. die Einengung des Spielfeldes. Aus der persönlichen Spielerfahrung

weiß der Autor aber zu berichten, dass gerade die bewusste Einengung der Spielfeldgröße elementar für eine gute Taktik ist. Dadurch werden die Gegner stark unter Druck gesetzt.

Beide Ansätze müssen miteinander verglichen werden. Der folgenden Tabelle sind die verschiedenen Bewertungsfunktionen zu entnehmen, die gegeneinander getestet werden:

Tabelle 5.2: Die zu vergleichenden Bewertungsfunktionen

Name	Berechnung	Kommentar
countPawns	p	Die Rollen der Figuren werden ignoriert
countPawns10Roles	$10 \cdot p + r$	Die Rollen werden verhältnismäßig stark gewichtet
countPawns100Roles	$100 \cdot p + r$	Die Anzahl der Figuren wird am stärksten gewichtet
countPawns100Moves	$100 \cdot p + m$	Die Rollengewichtung erfolgt über die Anzahl der möglichen Züge einer Figur

p ..Anzahl der Figuren,

r ..Punkte für die Rollen nach der vorherigen Tabelle,

m ..Anzahl der möglichen Züge der Figuren

Insgesamt werden vier Versionen gegeneinander antreten. Die Anzahl der Figuren ist bei allen die wichtigste Metrik. `countPawns` wird dabei die Basis-Version darstellen, da hier die Rollen der Figuren gar nicht mit bewertet werden. Zwei verschiedene Versionen der Rollengewichtung nach der Punktetabelle sind mit `countPawns10Roles` und `countPawns100Roles` vertreten. In der ersten Version wird die Anzahl der Spielfiguren mit dem Faktor 10 gewichtet. Da zwei Damen auch insgesamt 10 Punkte geben, erhält die Rollengewichtung einen stärkeren Einfluss auf das Ergebnis. Bei der zweiten Version ist der Faktor 100. Dadurch ist die Anzahl der Figuren stets die wichtigste Metrik und die Rollengewichtung erhält eine relativ kleine Bedeutung. Der alternative Ansatz, wo die Anzahl der Züge der Figuren gezählt werden, findet sich im `countPawns100Moves` wieder. Die Figurenanzahl wird hier ebenfalls mit 100 multipliziert.

Ein letzter Punkt ist noch wichtig zu beachten: Alle Versionen der Bewertungsfunktion gewichten die Spieler einzeln und getrennt von einander. Allerdings ist nicht nur wichtig, wie gut ein einzelner Spieler gerade dasteht. Viel mehr interessiert, wie ein Spieler *im Verhältnis zu den anderen* abschneidet. Um dies zu bewerkstelligen, wird die im vorigen Kapitel angesprochene *Normierung* verwendet. Dadurch sind die Bewertungen nicht länger von einander getrennt, sondern stehen im unmittelbaren Zusammenhang. Verbessert ein Spieler seine eigene Bewertung, sinkt automatisch die Bewertung der anderen Spieler. Dadurch ist der strikt-kompetitive Charakter des Spieles wiederhergestellt.

Algorithmen

Allgemein

Alle Algorithmen sind in einer rekursiven Form implementiert worden. Dies liefert u.U. nicht die besten Performance. Da JavaScript keine rein funktionale Programmiersprache ist, werden kaum Optimierungen von rekursiven Konstrukten vorgenommen. Die rekursive Definition ist aber deutlich einfacher zu implementieren gewesen. Da alle Algorithmen auf diese Art geschrieben sind, dürfte die Vergleichbarkeit dennoch gegeben sein.

Zusätzlich müssen alle Algorithmen ein gewisses Rahmen-Konstrukt implementieren, damit sie im Zusammenhang mit dem eigens für dieses Spiel entwickelte Session-Framework verwendet werden können. Auf diese Art und Weise ist der meiste Code für alle Algorithmen gleich und nur die spezifischen Details ändern sich.

Die Bewertungsfunktion ist dabei komplett parametrisiert. Sie wird also separat definiert und kann dann von jedem Algorithmus beliebig verwendet werden. Dies erzeugt einen hohen

Grad an Modularität und Flexibilität. Jede beliebige Bewertungsfunktion kann mit jedem beliebigen Algorithmus kombiniert werden.

Max^N

Der Max^N ist die Implementierung des Algorithmus von Luckhart und Irani (1986). In dieser Implementierung wird kein Pruning verwendet. Dieser Algorithmus entspricht am exaktesten dem Vorgehen aus der Spieltheorie. Es ist also der reinste Algorithmus in Bezug auf die Validität seiner Ergebnisse.

Er wird zum Vergleich mit den anderen Algorithmen dienen. Es müsste sich um den langsamsten Algorithmus handeln. Außerdem wird er zum Einsatz kommen, um die verschiedenen Bewertungsfunktionen miteinander zu vergleichen. Dadurch hat keine Berechnung einen Vorteil durch das Pruning.

Dieser Algorithmus verfügt über keinen eigenen Mechanismus, um die Bewertungen der Spieler gegeneinander zu vergleichen. Deswegen werden stets die normierten Ergebnisse der Bewertungsfunktion verwendet.

Der folgende Pseudocode basiert auf der Darstellung des Max^N aus dem Paper von Mikael Fridenfalk (2014). Alle diese Pseudocodes zeigen, wie algorithmisch die Auszahlung für einen bestimmten Spielbrettzustand (gs) mit einer vorgegebenen Tiefe ($depth$) ermittelt wird.

```

MAXN(gs, depth)
  if isGameOver(gs) or depth ≤ 0
    return normalize(eval(gs))
  bestScore = 0
  foreach nextGS
    score = MaxN(nextGS, depth - 1)
    if bestScore < score
      bestScore = score
  return bestScore

```

Max^{NIS}

Diese Implementierung verwendet das Immediate und Shallow Pruning im Max^N-Algorithmus. Dazu wird immer die aktuell beste Bewertung des Vaters an den Kindknoten übergeben (beim rekursiven Aufruf). Diese Variante müsste vor allem gegen Ende des Spieles tiefer suchen können als der normale Max^N und dadurch bessere Ergebnisse erzielen.

Auch dieser Algorithmus verwendet stets die normierten Ergebnisse der Bewertungsfunktion.

Der folgende Pseudocode entspricht im Wesentlichen dem Max^N, enthält allerdings kleine Modifizierungen, um das Immediate und Shallow Pruning umzusetzen.

```

MAXNIS(gs, depth, parentBestScore)
  if isGameOver(gs) or depth ≤ 0
    return normalize(eval(gs))
  maxScore = MAXSUM - parentBestScore
  bestScore = 0
  foreach nextGS
    score = MaxNIS(nextGS, depth - 1, bestScore)
    if bestScore < score
      bestScore = score
    if bestScore ≥ maxScore      // pruning condition
      break
  return bestScore

```

Die *MaxSum* beträgt 1. Dies ist schließlich der höchste mögliche Wert, den eine Auszahlung nach der Normierung annehmen kann.

Paranoid

Der Paranoid ist der “paranoide” Ansatz von Sturtevant und Korf (2000). Alle Gegner werden hier als ein Team betrachtet, die zusammen gegen den aktuellen Spieler spielen. Dadurch ist der Minimax-Algorithmus mit Alpha-Beta Pruning anwendbar. Dies ist das effektivste Pruning Verfahren, also müsste dieser Algorithmus theoretisch die höchste Suchtiefe erreichen.

Die Implementierung erfolgt so, dass beim ersten Aufruf ein Parameter gesetzt wird. Dieser Parameter enthält den “Spieler Max”, also den, der gegen das “Team Min” spielt. Die Bewertungsfunktionen werden ganz normal verwendet, wie bei den anderen Algorithmen auch. Sie gibt also einen Auszahlungsvektor zurück. Allerdings werden nun die Ergebnisse von Team Min von dem Ergebnis des Spielers Max abgezogen. Es ist also ein zusätzlicher Rechenschritt nötig. Dies sollte aber nicht zu stark ins Gewicht fallen.

Dieser Algorithmus gewichtet ja die einzelnen Bewertungen der Spieler automatisch gegeneinander, indem er sie von einander abzieht. Dadurch könnte die vorherige Normierung der Ergebnisse überflüssig sein. Die Normierung kann mittels eines Flags an- und ausgeschaltet werden. Es werden beide Versionen gegeneinander getestet und die bessere Variante wird anschließend gegen die anderen Algorithmen antreten.

Der folgende Pseudocode basiert auf der Darstellung des Minimax mit Alpha-Beta Pruning aus dem Paper von Mikael Fridenfalk (2014). Allerdings sind die speziellen Erweiterungen für “Chamäleon Schach” hinzugefügt worden.

```

PARANOID(gs, depth, maxPlayer,  $\alpha$ ,  $\beta$ , normalize)
    if isGameOver(gs) or depth ≤ 0
        scōre = eval(gs)
        if normalize
            scōre = normalize(scōre)
        return  $2 \cdot scōre_{maxPlayer} - sum(scōre)$ 
    foreach nextGS
        score = Paranoid(nextGS, depth - 1, maxPlayer,  $\alpha$ ,  $\beta$ , normalize)
        if gsplayer == maxPlayer
             $\alpha = max(\alpha, score)$ 
        else
             $\beta = min(\beta, score)$ 
        if  $\alpha \geq \beta$  // pruning condition
            break
        if gsplayer == maxPlayer
            return  $\alpha$ 
        else
            return  $\beta$ 
    
```

Hypermax

Der Hypermax ist der Algorithmus nach Fridenfalk (2014).

Für “Chamäleon Schach” sind ein paar kleine Modifizierungen nötig gewesen. Bei der Initialisierung des $\vec{\alpha}$, der die besten erreichten Bewertungen der jeweiligen Spieler speichert, dürfen nur die Spieler mit $-\infty$ initialisiert werden, die tatsächlich noch mitspielen. Ansonsten bleiben die Werte der Spieler, die nicht mehr dabei sind, auf $-\infty$ stehen und die Pruning-Bedingung wird niemals erreicht. Daher wird bei der Initialisierung ein Array der noch lebenden Spieler erzeugt und stets als Parameter mitgegeben.

Die Ergebnisse der Bewertungsfunktion werden ja mittels eines speziellen Verfahrens in den “Zero-Space” transferiert. Dadurch werden die Bewertungen bereits gegeneinander gewichtet, sodass eine Normierung nicht mehr nötig ist. Fridenfalk sagt selbst, dass dennoch gerne

ausprobiert werden kann, ob sich die Ergebnisse durch eine Normierung verbessern. Daher kann die Normierung mittels eines Flags an- und ausgeschaltet werden. Ähnlich wie beim Paranoid werden auch hier zuerst diese beiden Versionen gegeneinander antreten und die bessere dann gegen die anderen Algorithmen.

Der folgende Pseudocode basiert auf dem Paper von Mikael Fridenfalk (2014). Allerdings sind die speziellen Erweiterungen für “Chamäleon Schach” hinzugefügt worden.

```

HYPERMAX(gs, depth,  $\vec{\alpha}$ , players, normalize)
  if isGameOver(gs) or depth ≤ 0
    score = eval(gs)
    if normalize
      score = normalize(score)
    return scoreplayers - avg(scoreplayers)
  bestScore = -∞
  foreach nextGS
    score = Hypermax(nextGS, depth - 1,  $\vec{\alpha}$ , players, normalize)
    if  $\vec{\alpha}_{player} < score_{player}$ 
       $\vec{\alpha}_{player} = score_{player}$ 
      bestScore = score
    if sum( $\vec{\alpha}$ ) ≥ 0      // pruning condition
      break
  return bestScore

```

Session-Framework

Um die Vergleichbarkeit der Algorithmen zu gewährleisten, sollten die Algorithmen sich so viel Code wie möglich teilen. Nur die spezifischen Unterschiede in der Berechnung der Auszahlung werden separat implementiert.

Dazu ist für “Chamäleon Schach” ein Rahmen-Konstrukt entworfen, entwickelt und implementiert worden. Dieses Konstrukt (welches im Folgenden als Session-Framework bezeichnet wird) umfasst eine Algorithmus-Factory, mit der einheitliche Instanzen von den Algorithmen erzeugt werden können. Weiterhin definiert das Framework eine sog. Session. Diese Sessions werden verwendet, um die Algorithmus-Instanzen gegeneinander antreten zu lassen.

Algorithmus-Factory

Die Algorithmus-Factory erzeugt spezifische Algorithmus-Instanzen. Alle erzeugten Instanzen haben die gleiche Signatur: `algorithm: (gs, mode, modeValue) => (gs', depth, time)`.

Das bedeutet, dass eine Algorithmus-Instanz einen Spielbrettzustand `gs` übergeben bekommt. Zusätzlich noch den gewünschten Modus `mode` für die Berechnung sowie den Wert für den jeweiligen Modus `modeValue`. Als Ergebnis liefert die Berechnung den nächsten Spielbrettzustand `gs'`. Dies ist also der Spielbrettzustand, der entsteht, wenn der beste Zug (nach der Berechnung des Algorithmus) durchgeführt wird. `depth` und `time` sind die erreichte Suchtiefe der Berechnung sowie die benötigte Rechenzeit.

Eine Algorithmus-Instanz kann in zwei verschiedenen Modi arbeiten: `depth` oder `time`. Im `depth`-Modus wird der Spielbaum nur bis zu der festgelegten Tiefe aufgebaut. Die benötigte Rechenzeit spielt keine Rolle. Im `time`-Modus wird zunächst der Spielbaum auf der ersten Ebene aufgebaut und bewertet. Steht dann noch Rechenzeit zur Verfügung, so wird der Spielbaum um eine Ebene erweitert und wieder bewertet usw. Die verfügbare Rechenzeit wird dabei in Millisekunden definiert. Dieser Modus ist also die Implementierung des Iterative Deepening.

Die Algorithmus-Factory definiert einen festen Rahmen. Um Algorithmus-Instanzen erzeugen zu können, muss jeder Algorithmus ein spezielles Interface implementieren. Dann können

damit Instanzen erstellt werden.

Das Interface erwartet zunächst die Definition von drei generischen Datentypen:

- **S**: das genaue Format des Bewertungsergebnisses. Die meisten Algorithmen verwenden für die Bewertung den besagten Ergebnisvektor mit einem Eintrag je Spieler. Der Paranoid arbeitet allerdings mit skalaren Bewertungsergebnissen. Daher muss hier der verwendete Typ definiert werden.
- **A**: zusätzliche Daten. Wie zuvor erwähnt, brauchen die meisten Algorithmen zusätzliche Daten für die Berechnung. Der Paranoid braucht die Information, wer Spieler Max ist und die aktuellen α - und β -Werte. Der Hypermax braucht die Information, welche Spieler überhaupt noch mitspielen usw. In dieser generischen Datenstruktur können also zusätzliche Informationen für die Algorithmen definiert werden.
- **P**: generischer Start-Parameter. Hiermit können zusätzliche Parameter für die Initialisierung der Algorithmus-Instanz definiert werden. Er wird verwendet, um beim Paranoid und beim Hypermax die Normierung an- und ausschalten zu können.

Darüber hinaus müssen vier Funktionen implementiert werden:

- **init**: $(gs, P) \Rightarrow A$. Die Init-Funktion bekommt den aktuellen Spielzustand **gs** und den generischen Parameter **P** übergeben. Als Ergebnis gibt die Funktion ein Objekt vom generischen Typ **A** zurück. Mit dieser Funktion werden also die initialen Werte für die zusätzlichen Daten **A** gesetzt. Dies sind z.B. die noch mitspielenden Spieler für den Hypermax oder die α - und β -Werte für den Paranoid etc.
- **evalGameState**: $(gs, d, ef, A) \Rightarrow (S, A)$ ist die Kernfunktion. Hiermit werden die möglichen Spielzustände bewertet. Dazu wird der zu bewertende Spielzustand **gs**, die geforderte Suchtiefe **d**, die Bewertungsfunktion **ef** und das generische Objekt **A** an die Funktion übergeben. Als Ergebnis entsteht ein Tupel aus der Bewertung für den Zustand **S** und einer eventuell modifizierten Version der zusätzlichen Daten **A**.
- **onNextDepth**: $A \Rightarrow A$ ist ein Hook für den **time**-Modus. Bei manchen Algorithmen müssen die Daten in **A** angepasst werden, wenn die nächst höhere Suchtiefe erreicht wird. Beispielsweise müssen die α - und β -Werte für den Paranoid wieder erneut initialisiert werden usw. Diese Funktion wird also aufgerufen, wenn die nächste Suchtiefe erreicht wird und sie dient der Modifizierung von **A**.
- **findBestScoreIndex**: $(S[], A) \Rightarrow int$ wird ganz am Ende der Berechnung aufgerufen. Es muss ermittelt werden, welcher Folgespielzustand derjenige mit der besten Bewertung ist. Da die Bewertungen ja entweder Skalare oder Vektoren sind (abhängig von Typ **S**), muss diese Funktion definiert werden, um entsprechend die beste Bewertung finden zu können.

Sobald ein Algorithmus dieses Interface implementiert, kann zusammen mit der Implementierung und einer der Bewertungsfunktionen eine Algorithmus-Instanz erzeugt werden.

Session

Mit den Algorithmus-Instanzen kann nun eine Session durchgeführt werden. Eine Session erwartet eine Liste aus Algorithmus-Instanzen, den Modus (**depth** oder **time**) und den Modus-Wert. Anschließend spielt sie mehrere Spiele mit verschiedenen Anordnungen der Algorithmen um das Spielbrett. Der komplette Spielverlauf sowie die erreichten Suchtiefen und Rechenzeiten werden getrackt und abgespeichert.

Hintergrund: Bis zu vier Leute können bei "Chamäleon Schach" mitspielen. Spieler Rot ist dabei immer der Startspieler, also derjenige, der als erster seinen Zug ausführt. Danach folgen Spieler Blau, Gelb und Grün. Diese Reihenfolge ist immer so festgelegt. Es ist gut möglich, dass die Startposition einen Einfluss auf das Spielergebnis hat. U.U. gewinnt ein Spieler auf einer bestimmten Position häufiger oder seltener. Deswegen ist es wichtig, dass die Algorithmen mehrere Spiele in verschiedenen Anordnungen gegeneinander spielen.

Die Session geht nun so vor, dass sie alle möglichen Anordnungen (Permutationen) der Algorithmen um das Spielbrett generiert. Die genaue Anzahl an Permutationen ist davon abhängig, wieviele Algorithmen an die Session übergeben worden sind:

- Bei vier Algorithmen gibt es 24 verschiedene Möglichkeiten die Algorithmen auf die verschiedenen Startpositionen zu verteilen. Es werden ausschließlich Vier-Spieler-Spiele gespielt.
- Bei drei Algorithmen werden sowohl alle Permutationen für Drei- als auch für Vier-Spieler-Spiele generiert und gespielt. Die Vier-Spieler-Spiele laufen dabei so ab, dass immer jeweils einer der Algorithmen zwei Spielerpositionen übernimmt. Der Algorithmus spielt dann also nicht nur gegen die anderen, sondern auch gegen eine Instanz von sich selbst. Es werden nur Spiele mit klar verschiedenen Anordnungen der Algorithmen gespielt. Insgesamt werden hier 54 Spiele durchgeführt.
- Bei zwei Algorithmen werden sowohl Zwei-, Drei- also auch Vier-Spieler-Spiele gespielt. In den Drei- und Vier-Spieler-Spielen werden also Spielerpositionen doppelt bzw. auch dreimal mit dem selben Algorithmus besetzt. Wenn mehrere Anordnungen identisch zueinander sind, wird nur ein Spiel mit der jeweiligen Konstellation gespielt. Dadurch liegt die Zahl an durchgeführten Spielen hier bei 38.

Nach der Generierung der verschiedenen Permutationen, werden die Spiele in den entsprechenden Anordnungen gespielt. Die Ergebnisse werden in JSON-Dateien gespeichert. Dadurch stehen die Daten aus der Session für weitere spätere Analysen bereit. Die genaue Projektstruktur findet sich im angehängten Git-Repository.

Während der Versuche ist ein Phänomen aufgetreten, welches es so noch nicht gegeben hat: Die Algorithmen haben gegen Ende des Spieles eine Konstellation gefunden, in der sie ewig weiterspielen können. Sie haben im Spielbaum einen unendlichen Pfad entdeckt. Auf diesem Pfad geht das Spiel endlos weiter und keiner der Algorithmen kann den anderen in die Enge treiben und so einen Sieg erringen. Es ist praktisch für beide Algorithmen stets möglich, eine Niederlage zu vermeiden.

Im Spiel mit realen Menschen ist eine solche Situation noch nie vorgekommen. Entsprechend ist in den originalen Spielregeln auch kein Unentschieden vorgesehen gewesen. Die Algorithmen sind aber in der Lage, eine solche Konstellation zu finden. Daher ist nun eine neue Regel definiert worden: Ein Spiel endet mit einem Unentschieden für die noch lebenden Spieler, wenn nach dem 100. Zug noch immer kein Gewinner feststeht. Auf diese Art und Weise werden unendlich lange Spiele vermieden.

Anhand der Daten aus einer Session können nun Tabellen generiert werden, welche die Ergebnisse der Session zusammenfassen. Diese sind für die Auswertung verwendet worden. Die Ergebnisse geben zu jedem Algorithmus die Anzahl an Siegen, Niederlagen und Unentschieden wieder. Außerdem die erreichte Suchtiefe und die benötigte Rechenzeit. Beide Metriken werden sowohl im Median als auch im Durchschnitt berechnet. Mehr dazu im nächsten Kapitel.

6 Auswertung

Nun folgt der finale Schritt, in dem alle Algorithmen gegeneinander antreten und der beste ermittelt wird.

Bewertungsfunktionen

Als erstes gilt es die beste Bewertungsfunktion zu finden. Diese wird dann im Folgenden für alle Algorithmen verwendet. Dazu sind vier Instanzen des Max^N-Algorithmus erzeugt worden, jede mit einer anderen Bewertungsfunktion. Da das Session-Framework auch vier Algorithmen gleichzeitig gegeneinander antreten lassen kann, werden alle vier Instanzen gleichzeitig gegeneinander spielen.

Um die Effektivität der Bewertungsfunktionen miteinander vergleichen zu können, sollten die Algorithmen eine feste Suchtiefe haben. Tendenziell werden die Algorithmen ja umso stärker, je weiter sie den Spielbaum aufbauen und in die Zukunft schauen können. Daher ist die Vergleichbarkeit nur bei einer statischen Suchtiefe gegeben.

Es sind insgesamt vier Sessions gespielt worden. Die vorgegebene feste Suchtiefe betrug eins, zwei, drei und vier. Die folgende Tabelle fasst die Ergebnisse dieser vier Sessions zusammen:

Tabelle 6.1: Vergleich der Bewertungsfunktionen mit fester Tiefe

Algorithmus	Siege	Unentschieden	Niederlagen	Rechenzeit in ms
countPawns	13	0	83	659
countPawns10Roles	29	4	63	503
countPawns100Roles	29	3	64	523
countPawns100Moves	19	6	71	1.857

Da vier Algorithmen gegeneinander angetreten sind, wurden 24 Spiele pro Session gespielt. In der Tabelle ist für jede Bewertungsfunktion aufgelistet, wie häufig diese gewonnen, verloren oder unentschieden gespielt hat. Außerdem ist die durchschnittliche Rechenzeit angegeben.

Die Anzahl an gewonnenen, verlorenen und unentschieden geendeten Spielen für eine Bewertungsfunktion ist der wichtigste Indikator dafür, wie gut die jeweilige Funktion ist. Je besser die Qualität der vorhergesagten Auszahlung, desto besser spielt ein Algorithmus. Eine überlegene Bewertungsfunktion müsste bei gleicher Suchtiefe bessere Vorhersagen liefern. Dadurch trifft der Algorithmus bessere Entscheidungen und gewinnt häufiger. Die Anzahl der Siege ist damit die zentrale Metrik, um zu bestimmen, welche Bewertungsfunktion die beste ist.

Es fällt direkt auf, dass der `countPawns` am schlechtesten gespielt hat. Er hat die wenigsten Siege erringen können und am häufigsten verloren. Diese Bewertungsfunktion bestimmt die Auszahlung anhand der Anzahl der Figuren je Spieler. Die Rollen der Figuren werden nicht betrachtet. Allerdings ist ja gerade ein Hauptmerkmal von "Chamäleon Schach", dass sich die Rollen der Figuren ständig ändern. Insofern ist nicht verwunderlich, dass diese Bewertungsfunktion den anderen unterlegen ist. Alle anderen lassen zusätzlich noch die Rolle einer Figur mit in die Bewertung einfließen.

Eine zweite sehr erstaunliche Erkenntnis ist ebenfalls zu sehen: Der `countPawns100Moves` braucht im Schnitt die drei- bis vierfache Rechenzeit. Die anderen Bewertungsfunktionen brauchen zwischen 503 und 659 Millisekunden; `countPawns100Moves` liegt bei 1.857 Millisekunden. Bei dieser Funktion wird zu der Anzahl der Figuren je Spieler zusätzlich noch gezählt,

wieviele mögliche Züge die jeweiligen Figuren aktuell ausführen können. Anscheinend ist dies ein sehr aufwendiger Vorgang. Da diese Bewertungsfunktion auch nur recht mittelmäßig abgeschnitten hat, wird sie ebenfalls nicht weiter verwendet werden. Es ist wichtig, dass die Algorithmen so viele Berechnungen wie möglich in der kurzen verfügbaren Zeit durchführen. Da ist eine langsame Bewertungsfunktion nicht brauchbar.

Damit bleiben noch `countPawns10Roles` und `countPawns100Roles` übrig. Beide führen eine Rollengewichtung anhand von festen Punkten je Rolle durch (siehe vorheriges Kapitel). Der Unterschied liegt darin, wie stark dazu die Anzahl der Figuren je Spieler gewichtet wird. Beide Funktionen liefern in etwa die gleiche Performance. Beide haben jeweils 29 mal gewonnen, der `countPawns10Roles` hat einmal mehr unentschieden gespielt.

Um einen klaren Sieger zu ermitteln, wird nun der Vorgang nur mit diesen beiden Funktionen wiederholt. Das heißt, es werden wieder vier Sessions gespielt. Dieses mal aber nur mit den beiden verbleibenden Bewertungsfunktionen. Die Ergebnisse sind in der folgenden Tabelle festgehalten:

Tabelle 6.2: Vergleich der Bewertungsfunktionen mit Rollengewichtung mit fester Tiefe

Algorithmus	Siege	Unentschieden	Niederlagen	Rechenzeit in ms
<code>countPawns10Roles</code>	71	4	77	607
<code>countPawns100Roles</code>	77	4	71	595

Im direkten Vergleich gegeneinander scheint der `countPawns100Roles` ein wenig besser abzuschneiden. Er hat 77 Spiele gewonnen, sein Kontrahent nur 71. Dieses Ergebnis deckt sich mit der Spielerfahrung des Autors: Der `countPawns100Roles` gewichtet die Anzahl der Figuren je Spieler deutlich stärker als sein Bruder. Dadurch ist es zwar auch wichtig, die eigenen Figuren möglichst gut zu positionieren, am wichtigsten ist es aber für die Funktion, keine Figuren zu verlieren. Der Autor kann bestätigen, dass es im Spiel am wichtigsten ist, dafür zu sorgen, dass keine eigenen Figuren geschlagen werden. Das ist viel wichtiger als eine starke Rolle inne zu haben.

Damit ist die Bewertungsfunktion für “Chamäleon Schach” gefunden: es ist der `countPawns100Roles`.

Paranoid - normiert oder nicht

Bevor nun die verschiedenen Algorithmen gegeneinander antreten können, müssen noch zwei Fragen geklärt werden: Sowohl der Paranoid als auch der Hypermax verfügen beide über einen internen Mechanismus, um die Bewertungen der einzelnen Spieler gegeneinander zu gewichten. Daher ist die Normierung der Ergebnisse der Bewertungsfunktion (wie sie beim Max^N und beim Max^NIS verwendet wird) eventuell obsolet.

Um festzustellen, welche Version verwendet werden soll, treten nun die Algorithmen einmal in der normierten und einmal in der nicht normierten Form gegeneinander an. Wie zuvor ist die Anzahl der gewonnenen Spiele die entscheidende Metrik. Sie zeigt, wie gut die berechneten Auszahlungen sind, auf deren Grundlage die Entscheidungen getroffen werden.

Als erstes wird der Paranoid betrachtet. Es sind vier Sessions mit konstanter Suchtiefe von eins, zwei, drei und vier gespielt worden. Das sind die Ergebnisse:

Tabelle 6.3: Vergleich des Paranoid mit und ohne Normierung mit fester Suchtiefe

Algorithmus	Siege	Unentschieden	Niederlagen	Rechenzeit in ms
n. normiert	70	5	77	224
normiert	77	5	70	228

Die normierte Version hat etwas besser abgeschnitten. Sie hat 77 Siege erringen können, die nicht normierte nur 70. Die Rechenzeit unterschiedet sich kaum. Die Normierung ist aber ein zusätzlicher Rechenschritt, daher benötigt die normierte Version vier Millisekunden länger.

Trotzdem soll nun sichergestellt werden, dass der Unterschied in der Rechenzeit wirklich keinen Einfluss auf das Ergebnis hat. Deswegen sind weitere drei Sessions gespielt worden: dieses mal mit einer vorgegebenen Rechenzeit von 10, 100 und 1.000 Millisekunden. Das sind die Ergebnisse:

Tabelle 6.4: Vergleich des Paranoid mit und ohne Normierung mit fester Rechenzeit

Algorithmus	Siege	Unentschieden	Niederlagen	Suchtiefe (Median)
n. normiert	45	3	66	3,50
normiert	66	3	45	3,46

In diesem Modus ist die normierte Version sogar noch stärker als die nicht normierte. 66 Siege der normierten zu 45 Siegen der nicht normierten Version sprechen für sich. Durch den zusätzlichen Rechenschritt schafft die normierte Version nicht die gleiche Suchtiefe, aber die Werte liegen fast gleich auf. Warum zum Vergleich der Suchtiefe der Median verwendet wird, wird später erklärt.

Der Unterschied in der Performance lässt sich folgendermaßen erklären: In der nicht normierten Version werden die *isolierten Einzelbewertungen* der Spieler gegeneinander gerechnet. Die normierte Version hingegen gewichtet die *Gewinnwahrscheinlichkeiten* der Spieler gegeneinander. Die Gewinnwahrscheinlichkeiten sind in ihrem Charakter deutlich kompetitiver. Deswegen spielt diese Version des Paranoid besser.

Der Paranoid wird also im **normierten** Modus gegen die anderen Algorithmen antreten.

Hypermax - normiert oder nicht

Der gleiche Vorgang wie beim Paranoid wird nun für den Hypermax wiederholt. Zunächst also die Ergebnisse von vier Sessions mit fester Suchtiefe von eins, zwei, drei und vier:

Tabelle 6.5: Vergleich des Hypermax mit und ohne Normierung mit fester Tiefe

Algorithmus	Siege	Unentschieden	Niederlagen	Rechenzeit in ms
n. normiert	77	3	72	363
normiert	72	3	77	369

Für den Hypermax scheint die nicht normierte Version geringfügig stärker zu sein. Sie hat 77 mal gewonnen, die normierte Version nur 72 mal. Fridenfalk (2014) hat selbst angeführt, dass seine Transformation des Auszahlungsvektors in den “Zero-Space” vollkommen ausreichend ist, um die Auszahlungen gegeneinander zu gewichten. Die Normierung ist nur ein zusätzlicher Rechenschritt, der keinen Mehrwert liefert.

Zur Sicherheit werden noch drei weitere Sessions mit einer festen Rechenzeit von 10, 100 und 1.000 Millisekunden gespielt. Hier die Ergebnisse:

Tabelle 6.6: Vergleich des Hypermax mit und ohne Normierung mit fester Rechenzeit

Algorithmus	Siege	Unentschieden	Niederlagen	Suchtiefe (Median)
n. normiert	55	6	53	3,76
normiert	53	6	55	3,70

Auch hier scheint die nicht normierte Version leicht überlegen zu sein – sie hat zwei Spiele mehr gewonnen als die nicht normierte. Allerdings sind sechs Spiele unentschieden ausgegangen. Die Normierung hat also faktisch keinen wirklichen Einfluss auf die Performance des Hypermax. Die erreichte Suchtiefe ist aber etwas geringer mit Normierung aufgrund des zusätzlichen Rechenschritts, der stets durchgeführt werden muss.

Der Hypermax wird also **ohne Normierung** arbeiten.

Bester Algorithmus für Chamäleon Schach

Endlich folgt die finale Auswertung. Es gilt, den besten Algorithmus für “Chamäleon Schach” zu finden. Wie zuvor, wird die Performance eines Algorithmus daran festgemacht, wie häufig er im Vergleich zu den anderen gewinnt.

Da insgesamt vier Algorithmen implementiert worden sind und “Chamäleon Schach” zu viert gespielt werden kann, treten alle vier gleichzeitig gegeneinander an. Als erstes sind vier Sessions mit fester Suchtiefe gespielt worden. Wie üblich waren die Suchtiefen eins, zwei, drei und vier. Die folgende Tabelle zeigt die Ergebnisse der vier Sessions:

Tabelle 6.7: Vergleich der vier Algorithmen mit fester Suchtiefe

Algorithmus	Siege	Unentschieden	Niederlagen	Rechenzeit in ms
— Suchtiefe: 1				
Max ^N	6	0	18	< 1
Max ^N IS	6	0	18	< 1
Hypermax	6	0	18	< 1
Paranoid	6	0	18	< 1
— Suchtiefe: 2				
Max ^N	5	4	15	2
Max ^N IS	5	4	15	2
Hypermax	4	4	16	2
Paranoid	4	0	20	3
— Suchtiefe: 3				
Max ^N	8	0	16	70
Max ^N IS	8	0	16	69
Hypermax	8	0	16	69
Paranoid	0	0	24	33
— Suchtiefe: 4				
Max ^N	7	0	17	2.524
Max ^N IS	7	0	17	2.513
Hypermax	4	0	20	2.637
Paranoid	6	0	18	651

Mit einer Suchtiefe von eins spielen alle Algorithmen gleich gut. Dies ist nicht verwunderlich, da hier keine Rückwärtsauflösung und kein Pruning stattfindet. Da alle Algorithmen die gleiche Bewertungsfunktion verwenden, spielen sie alle komplett gleich.

Mit steigender Suchtiefe schneiden der Paranoid und der Hypermax immer schlechter im Vergleich zu den Max^N-Algorithmen ab. Auch dieses Ergebnis ist zu erwarten gewesen. Der Hypermax beschneidet teilweise zu radikal den Spielbaum und liefert damit schlechtere Auszahlungen. Der Paranoid geht von der falschen Annahme aus, alle Gegner hätten sich gegen den Spieler verbündet. Bei beiden Algorithmen ist klar, dass die Qualität ihrer Ergebnisse mit steigender Suchtiefe immer schlechter wird. Ihr Vorteil soll darin liegen, dass sie durch effektiveres Pruning eine höhere Suchtiefe in der gleichen Rechenzeit erreichen und dadurch bessere “weitsichtigere” Auszahlungen liefern. Da hier alle Algorithmen die gleiche Suchtiefe verwenden, schneiden sie entsprechend schlechter ab.

Es ist außerdem sehr deutlich zu sehen, wie effektiv das Alpha-Beta Pruning im Paranoid

funktioniert. Bei einer Suchtiefe von vier brauchen die anderen Algorithmen durchschnittlich circa 2.600 Millisekunden für die Berechnung. Der Paranoid ist mit 651 Millisekunden viel früher fertig.

Die Effektivität des Prunings im Hypermax ist nicht zu erkennen. Auf Suchtiefe vier ist er sogar der langsamste Algorithmus. Allerdings hat er da auch die meisten Spiele verloren. Geraade gegen Ende des Spieles gehen die Berechnungen sehr schnell, weil nur noch wenige Figuren übrig sind und die Spielbäume entsprechend kleiner werden. Das könnte der Grund sein, warum der Hypermax so "lange" braucht. Er hat einfach zu wenige "schnelle" Berechnungen aus dem Endspiel vorzuweisen.

Die entscheidende Frage ist aber, wie die Algorithmen gegeneinander abschneiden, wenn die Rechenzeit fest vorgegeben ist. Dies ist ja das Szenario, welches auch in der App verwendet wird. Außerdem können nun effektivere Pruning Verfahren den entscheidenden Unterschied liefern, da in der gleichen Zeit wesentlich mehr Züge im Spielbaum untersucht werden können. Daher sind nun drei Sessions mit fester Rechenzeit gespielt worden. Die verfügbaren Zeiten waren 10, 100 und 1.000 Millisekunden. Die folgende Tabelle zeigt die Ergebnisse:

Tabelle 6.8: Vergleich der vier Algorithmen mit fester Rechenzeit

Algorithmus	Siege	Unentschieden	Niederlagen	Suchtiefe (Durchschnitt)	Suchtiefe (Median)
<hr/>					
— Rechenzeit: 10ms					
Max ^N	3	2	19	3,58	2,50
Max ^N IS	3	3	18	4,42	3,14
Hypermax	3	1	20	4,11	2,32
Paranoid	9	6	9	4,55	4,09
— Rechenzeit: 100ms					
Max ^N	5	2	17	4,97	3,50
Max ^N IS	11	1	12	5,98	3,90
Hypermax	3	2	19	5,39	3,57
Paranoid	2	1	21	5,27	4,08
— Rechenzeit: 1000ms					
Max ^N	9	0	15	5,17	4,08
Max ^N IS	10	1	13	7,60	4,11
Hypermax	4	0	20	5,37	4,02
Paranoid	0	1	23	6,06	4,21

Die erreichte Suchtiefe ist in der Tabelle sowohl im Durchschnitt als auch im Median angegeben. Es ist deutlich zu sehen, dass der Durchschnitt stets wesentlich höher liegt als der Median. Außerdem ist die durchschnittliche Suchtiefe sehr "sprunghaft" (die Werte bei einer Rechenzeit von 1.000 Millisekunden sind sehr durchwachsen im Durchschnitt, im Median liegen sie viel näher beieinander). Der Hintergrund dazu ist, dass gegen Ende des Spieles die Spielbäume sehr klein werden, weil nur noch wenige Figuren übrig sind. Dadurch werden hier extrem hohe Suchtiefen erreicht. Ein Algorithmus, der sehr häufig gewinnt, hat dadurch sehr viele hohe Ausreißer in der Suchtiefe. Daher ist der Median der zuverlässigere Vergleichswert. Das ist auch der Grund, warum in den vorherigen Tabellen stets nur die Ergebnisse im Median angegeben worden sind.

Der Paranoid erreicht bei einer Rechenzeit von 10 Millisekunden eine sehr hohe Suchtiefe von 4,09. Damit liegt er viel höher als die anderen Algorithmen mit weitem Abstand. Diese hohe Suchtiefe ist mit Erfolg gekrönt. Er hat neun Spiele gewonnen und sechs unentschieden gespielt. Damit ist er der klare Sieger und es ist deutlich zu sehen, wie das effektive Pruning-Verfahren tatsächlich zu einer stärkeren Spielweise führt.

Dieser Erfolg ist allerdings nicht von Dauer. Bei einer Rechenzeit von 100 Millisekunden gewinnt der Paranoid nur noch zwei von 24 Spielen und bei einer Sekunde kann er gar keinen Sieg mehr vorweisen. In jedem Fall ist er der Algorithmus mit der höchsten erreichten Suchtiefe. Allerdings holen die anderen Algorithmen in Bezug auf die Suchtiefe sehr schnell

auf. Dadurch fällt die “falsche” Prämisse des Paranoid immer stärker ins Gewicht und seine Performance sinkt rapide.

Je höher die verfügbare Rechenzeit ist, desto mehr gleichen sich die erreichten Suchtiefen der Algorithmen einander an. Dies ist damit zu erklären, dass die Komplexität des Spielbaumes exponentiell wächst. Die Pruning-Verfahren bringen nur in bestimmten Fällen eine Verbesserung der Rechenzeit. Die Komplexität wächst aber schneller als der Effekt der verschiedenen Pruning-Verfahren. Mit höherer Rechenzeit sinken also die Unterschiede in der erreichten Suchtiefe der Algorithmen. Mit anderen Worten, je mehr Zeit zur Verfügung steht, desto unwichtiger wird das verwendete Pruning-Verfahren.

Dieses Phänomen ist sehr deutlich beim Paranoid zu sehen. Bei einer Rechenzeit von 10 Millisekunden erreicht er eine Suchtiefe von 4,09. Mit der hundertfachen Rechenzeit (von einer Sekunde) liegt die erreichte Suchtiefe “nur” bei 4,21. Das ist zwar trotzdem die höchste Suchtiefe im Vergleich zu den anderen Algorithmen, dennoch ist es verwunderlich, dass beide Werte so nah beieinander liegen. Es muss allerdings berücksichtigt werden, dass der Paranoid bei der langen Rechenzeit gar kein Spiel mehr gewonnen hat. Dadurch ist er nie in der Endphase des Spieles gewesen, wo viele hohe Ergebnisse in der Suchtiefe erzielt werden.

Würde die verfügbare Rechenzeit in der App nur sehr kurz sein, dann wäre der Paranoid die beste Wahl. Da laut Design-Konzept aber eine Sekunde “Bedenkzeit” zur Verfügung steht, wird der Paranoid nicht als Computergegner für “Chamäleon Schach” verwendet werden.

Die persönliche Spielerfahrung des Autors gibt einen Hinweis darauf, warum der Paranoid so schlecht abgeschnitten hat: Es ist eine recht geschickte Taktik, sich zu Beginn des Spieles etwas “einzumauern”. Man platziert seine Figuren so, dass sie sich alle gegenseitig decken. Dadurch haben die Gegner kaum Interesse, die eigenen Figuren zu schlagen. Nachdem sich die gegnerischen Spieler dann schon ein wenig gegenseitig “dezimiert” haben, wechselt man zu einer offensiveren Spielweise. Dieses Vorgehen hat sich schon öfter als sehr effektive Taktik erwiesen. Auf diese Spielweise würde der Paranoid allerdings nicht kommen, da er ja davon ausgeht, dass alle Gegner im Team zusammen spielen. Sie hätten demnach auch keinen Grund sich gegenseitig zu schlagen, aber genau das tun sie in Wirklichkeit rigoros. Daher ist der paranoide Ansatz in anderen Spielen sicher eine solide Lösung, für “Chamäleon Schach” macht er aber keinen Sinn – sofern genug Rechenzeit zur Verfügung steht.

Sehr enttäuschend ist das Ergebnis des Hypermax. Er hat in jeder Session nur drei oder vier Siege errungen. In den meisten Fällen ist er auch der Algorithmus mit der geringsten erreichten Suchtiefe. Das ist sehr erstaunlich, da der Max^N ja gar kein Pruning verwendet und trotzdem tiefer zu suchen scheint. Dies ist nur damit zu erklären, dass der Hypermax immer sehr früh im Spiel ausgeschieden ist. Gerade am Anfang sind viele Figuren auf dem Brett und die Spielbäume entsprechend komplex. Allerdings spricht etwas gegen diese Theorie: Seine durchschnittliche Suchtiefe liegt stets höher als die des Max^N. Es scheint also noch andere Einflüsse zu geben, die aber aus den erhobenen Daten nicht ersichtlich sind. Tatsächlich spielt das aber auch keine große Rolle. Der Hypermax wird nicht als Algorithmus für “Chamäleon Schach” verwendet werden.

Damit bleiben noch die Max^N-Algorithmen übrig. Sie lieferten beide sehr solide Ergebnisse. Ab 100 Millisekunden steht ihnen auch genügend Rechenzeit zur Verfügung, dass sie eine ordentliche Suchtiefe erreichen. Ab dieser Menge an verfügbarer Zeit sind sie die klaren Sieger und gewinnen praktisch jedes Spiel. Der Max^NIS kann durch die Pruning-Verfahren tatsächlich den Spielbaum schneller durchsuchen und erreicht stets eine höhere Suchtiefe als der Max^N ohne Pruning. Dadurch spielt er auch besser und gewinnt mehr Spiele. Bei einer Rechenzeit von einer Sekunde erreicht der Max^NIS sogar im Durchschnitt eine extrem hohe Suchtiefe von 7,6. Dieser Wert kommt vor allem dadurch zustande, dass der Max^NIS die meisten Spiele gewonnen hat und demnach viele hohe Ausreißer in der Suchtiefe aus dem Endspiel hat. Hinzu kommt, dass die Pruning-Verfahren Immediate and Shallow Pruning vorwiegend gegen Ende des Spieles ihre volle Effektivität entwickeln. Dadurch nehmen die hohen Ausreißer noch extremere Werte an.

Insgesamt ist damit der beste Algorithmus für “Chamäleon Schach” gefunden: Es ist der **Max^NIS**.

Der Vollständigkeit halber sind noch drei weitere Sessions gespielt worden. Hierbei sind die Algorithmen Max^NIS, Hypermax und Paranoid nochmal einzeln gegeneinander angetreten. Sie haben mit einer festen Rechenzeit von einer Sekunde gegeneinander gespielt. Die Ergebnisse sind in den folgenden drei Tabellen festgehalten:

Tabelle 6.9: Max^NIS vs Hypermax mit fester Rechenzeit von 1.000ms

Algorithmus	Siege	Unentschieden	Niederlagen	Suchtiefe (Durchschnitt)	Suchtiefe (Median)
Max ^N IS	20	4	14	11.08	4.56
Hypermax	14	4	20	9.09	5.09

Tabelle 6.10: Max^NIS vs Paranoid mit fester Rechenzeit von 1.000ms

Algorithmus	Siege	Unentschieden	Niederlagen	Suchtiefe (Durchschnitt)	Suchtiefe (Median)
Max ^N IS	21	0	17	7.58	4.11
Paranoid	17	0	21	5.90	4.27

Tabelle 6.11: Hypermax vs Paranoid mit fester Rechenzeit von 1.000ms

Algorithmus	Siege	Unentschieden	Niederlagen	Suchtiefe (Durchschnitt)	Suchtiefe (Median)
Hypermax	23	1	14	6.56	4.28
Paranoid	14	1	23	5.79	4.23

Die Ergebnisse bestätigen allerdings nur die vorherigen. Der Max^NIS ist stets der Algorithmus mit den meisten Siegen. Der Paranoid ist bei diesen langen Rechenzeiten stets der Verlierer.

Auch der Hypermax schneidet genauso ab wie zuvor, in Bezug auf Siege und Niederlagen. Allerdings gibt es in diesem Versuch mehrere Spiele, in denen der Hypermax auch gewonnen hat. Damit ist er häufiger im Endspiel und erreicht wesentlich bessere Ergebnisse in der erreichten Suchtiefe. Tatsächlich ist er hier immer der Algorithmus mit der höchsten Suchtiefe. Er übertrifft sogar den Paranoid.

Trotzdem bringen diese Erkenntnisse keine Änderungen im Ergebnis: Der Sieger bleibt der **Max^NIS**.

7 Fazit und Ausblick

Im Rahmen dieser Bachelor Arbeit ist ein Algorithmus für den Computergegner zu dem Brettspiel “Chamäleon Schach” entwickelt worden. Dabei wurde zunächst geklärt, was überhaupt eine “intelligente” Spielweise ist. Diese Frage hat in die Entscheidungstheorie bzw. im Speziellen in die Spieltheorie geführt. Hier ist theoretisch ermittelt worden, wie in einem Brettspiel wie “Chamäleon Schach” eine Entscheidung zu treffen ist. Dabei ist aber auch deutlich geworden, dass dieser theoretische Ansatz nicht umgesetzt werden kann. Das liegt an der hohen Komplexität des Spieles.

Daher führte der Weg nun zum heuristischen Ansatz, der in Grundzügen genauso arbeitet, wie die spieltheoretische Grundlage. Für die Lösung solcher Probleme gibt es eine Vielzahl von Algorithmen. Für Zwei-Spieler-Spiele gibt es einen Algorithmus, der allen anderen klar überlegen ist: der Minimax mit Alpha-Beta Pruning. Dieser gilt als Standard-Lösung. Ab drei Spielern gibt es aber keinen klar überlegenen Algorithmus mehr. Die einzige Möglichkeit herauszufinden, welcher am besten für das jeweilige Brettspiel geeignet ist, besteht im Testen.

Als nächstes sind verschiedene Algorithmen identifiziert worden, die für “Chamäleon Schach” infrage kommen. Dann wurden diese Algorithmen für das Spiel implementiert. Darüber hinaus ist ein Session-Framework entworfen und umgesetzt worden, welches die Algorithmen auf eine einheitliche Art und Weise gegeneinander antreten lassen kann.

Anschließend ist das Spiel analysiert worden, um verschiedene Ansätze für die heuristische Bewertungsfunktion zu finden. Diese sind dann in mehreren Versuchsreihen gegeneinander getestet worden. Dadurch konnte die beste Version dieser Bewertungsfunktion ermittelt werden. Weitere Tests folgten, in denen die speziellen Konfigurationen der Algorithmen gefunden worden sind (dabei ging es vorwiegend um die Normierung der Auszahlungsvektoren).

In einem letzten Schritt sind die verschiedenen Algorithmen gegeneinander angetreten, um den Sieger zu ermitteln: Gewonnen hat der Max^N-Algorithmus mit Immediate und Shallow Pruning. Dieser Algorithmus orientiert sich sehr genau an dem Vorgehen aus der Spieltheorie. Durch geschickte Pruning-Verfahren wird seine Rechenzeit verbessert.

Insgesamt steht den Algorithmen sehr viel Rechenzeit zur Verfügung aufgrund der Anforderungen aus dem App-Design. Eine Sekunde ist eine sehr lange Zeit. Da die Komplexität des Spielbaums exponentiell wächst und die Pruning-Verfahren für Spiele mit mehr als zwei Spielern alle nicht sehr effizient sind, gleichen sich die erreichten Suchtiefen aller Algorithmen immer weiter an, je mehr Rechenzeit zur Verfügung steht. Das Ergebnis ist, dass die Algorithmen am stärksten sind, die keine falschen Annahmen über das Spiel treffen (Paranoid) oder durch zu radikales Pruning wichtige Zweige im Spielbaum beschneiden (Hypermax).

Sicherlich hätte in dieser Arbeit noch so einiges mehr betrachtet und analysiert werden können: Beispielsweise könnten noch deutlich mehr Versionen der Bewertungsfunktion definiert und gegeneinander getestet werden. Ebenso gibt es noch eine ganze Reihe an weiteren verschiedenen Pruning-Verfahren zum Implementieren und Ausprobieren. Eine Optimierung der Performance der Algorithmen hätte auch andere Ergebnisse hervorbringen können. Sie sind alle mit rekursiven Funktionsaufrufen geschrieben und die verwendete Programmiersprache ist JavaScript gewesen. Beides ist nicht für eine herausragende Performance bekannt. Hier gibt es noch viele Stellschrauben, an denen gedreht werden könnte.

Ein Thema, welches sehr interessant für zukünftige Arbeiten wäre, ist der Hypermax-Algorithmus. Er liefert nicht die gleichen Ergebnisse wie der Max^N, weil sein Pruning zu rigoros vorgeht. Interessant wäre, genau zu analysieren, warum es zu diesen unterschiedlichen Ergebnissen kommt. Möglicherweise kann dieser Umstand durch eine leichte Modifikation des Algorithmus gelöst werden. Vielleicht ließe sich dadurch eine neue effektive Art des Prunings finden. Das könnte u.U. auch in anderen Bereichen außerhalb der Spieltheorie hilfreich sein bspw. beim Beschneiden von Entscheidungsbäumen.

Ein weiteres Thema, welches hier komplett ausgeblendet worden ist, ist die Performance der Algorithmen gegen echte Menschen. Der ermittelte Sieger-Algorithmus ist der beste im direkten Kampf mit anderen Algorithmen. Da es ja aber ein Computergegner für ein Spiel sein soll, welches von Menschen gespielt wird, wären in Zukunft Tests gegen menschliche Gegner angebracht.

Alternativ zum heuristischen Ansatz gibt es noch weitere Methoden zur Umsetzung eines Computergegners. Bspw. könnte ein neuronales Netz oder andere Arten von selbstlernenden Algorithmen ebenfalls gute Ergebnisse liefern. Dies ist ein Feld, welches deutlich den Rahmen dieser Arbeit gesprengt hätte, die Idee existiert allerdings.

Fürs erste ist aber ein starker Computergegner für “Chamäleon Schach” gefunden: der Max^N mit Immediate und Shallow Pruning.

Quellen

- „A008907 – OEIS“. o. J. Zugegriffen 25. Mai 2020. <https://oeis.org/A008907>.
- Abu Dalfaa, Mohaned, Bassem S. Abu-Nasser, und Samy S. Abu-Naser. 2019. „Tic-Tac-Toe Learning Using Artificial Neural Networks“.
- Dudenredaktion. o. J. „Heuristik – auf Duden online“. Zugegriffen 25. Mai 2020. <https://www.duden.de/node/66177/revision/66213>.
- Fridenfalk, Mikael. 2014. „N-Person Minimax and Alpha-Beta Pruning“. In *NICOGRAH International 2014*, 43–52.
- Geithner, Michael, und Martin Thiele. 2013. *Nachgemacht: Spieletipps aus der DDR*. DDR Museum Verlag.
- Großkopf, W., und L. Schubert. 2019. *Gesellschaftsspiele: Zum Mitmachen, Nachmachen und Selberbauen*. Selbstverlegt.
- Knuth, Donald E., und Ronald W. Moore. 1975. „An Analysis of Alpha-Beta Pruning“. *Artificial Intelligence* 6 (4): 293–326.
- Luce, Robert Duncan, und Howard Raiffa. 1957. *Games and Decisions: Introduction and Critical Survey*. John Wiley, New York. <https://www.buecher.de/40289426/>.
- Luckhart, Carol A., und Keki B. Irani. 1986. „An Algorithmic Solution of N-Person Games“. In *AAAI*, 86:158–62.
- Morgenstern, Oskar, und John von Neumann. 1947. *Theory of Games and Economic Behavior*. Princeton University Press.
- Pearl, J. 1984. „Heuristics Addison-Wesley“. *Reading, MA*.
- Riechmann, Thomas. 2014. *Spieltheorie*. Vahlen.
- Shannon, Claude E. 1950. „Programming a Computer for Playing Chess“. *Philosophical Magazine* 41 (314): 256–75.
- Sturtevant, Nathan R. 2003. „Multi-Player Games: Algorithms and Approaches“. Dissertation, University of California.
- Sturtevant, Nathan R., und Richard E. Korf. 2000. „On Pruning Techniques for Multi-Player Games“. *AAAI-2000* 49: 201–7.
- „Tic Tac Toe – Wikiludia“. o. J. Zugegriffen 25. Mai 2020. http://wikiludia.mathematik.uni-muenchen.de/wiki/index.php?title=Tic_Tac_Toe&oldid=8115.
- „universe atom count – WolframAlpha“. o. J. Zugegriffen 25. Mai 2020. <https://www.wolframalpha.com/input/?i=universe+atom+count>.

Anhang

Spielanleitung für Chamäleon Schach



Abbildung 7.1: Das Logo von Chamäleon Schach

Chamäleon Schach ist ein schachähnliches Brettspiel für zwei bis vier Personen.

Chamäleons sind kleine Echsen, die im Urwald leben und ihre Farbe an die Umgebung anpassen. Auf diese Weise tarnen sie sich vor ihren Fressfeinden.

Der besondere Clou bei diesem Spiel besteht darin, dass das Brett nicht nur aus schwarzen oder weißen Feldern besteht, sondern aus vier Farben. So gibt es rote, grüne, gelbe und blaue Felder.

Die Spielfiguren sind kleine Chamäleons. Je nachdem welche Farbe das Feld hat, auf dem eine Figur gerade steht, nimmt sie eine andere Rolle an. So ist sie vielleicht auf einem roten Feld ein Springer, auf einem grünen aber eine Dame.

Dadurch entsteht eine sehr spritzige und dynamische Variante des Schachspiels. Der Spielspaß ist garantiert. :-)

Spielziel

Jeder Spieler beginnt mit 4 Spielfiguren, die in schachähnlichen Zügen über das Brett bewegt werden und einander schlagen können. Wie sich eine Figur bewegen kann, ist abhängig von der Farbe des Feldes, auf dem sie gerade steht. Sobald ein Spieler keine Figuren mehr übrig hat (weil alle geschlagen worden sind), so hat dieser Spieler verloren und ist aus dem Spiel. Wer als letztes noch Figuren auf dem Brett hat, gewinnt.

Vorbereitung

Das Spielbrett mit 8x8 bunten Feldern wird in die Mitte gelegt. Die Felder des Spielbretts haben jeweils eine von vier Farben (rot, grün, gelb, blau). Die Anordnung der Farben ist vorgegeben.

Die Spieler wählen sich jeweils eine von insgesamt vier Farben (rot, grün, gelb oder blau). Je nach Farbe setzen sich die Spieler entsprechend um das Spielbrett. Dabei gehört immer die Seite zu einem Spieler, bei der das linke Eckfeld der Farbe des Spielers entspricht.

Nun nehmen die Spieler sich die vier Spielfiguren in ihrer Farbe und platzieren sie auf den vier linken Feldern, beginnend beim linken Eckfeld. Dabei sind die Figuren so zu setzen, dass alle Figuren zu Beginn die Rolle "Springer" haben.

Spielfiguren

Eine Figur gehört zu genau einem Spieler. Eine Figur nimmt ein ganzes Feld auf dem Brett ein. Es kann nur maximal eine Figur auf einem Feld stehen.

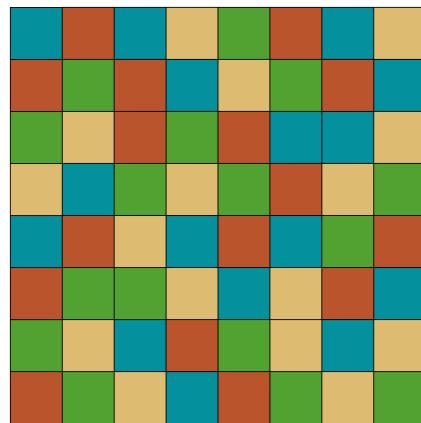


Abbildung 7.2: Das farbenfrohe Spielbrett

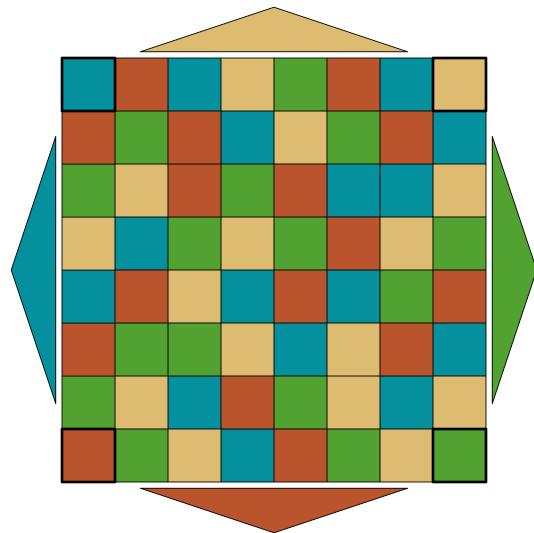


Abbildung 7.3: Das Spielbrett mit den Spielern an den Seiten

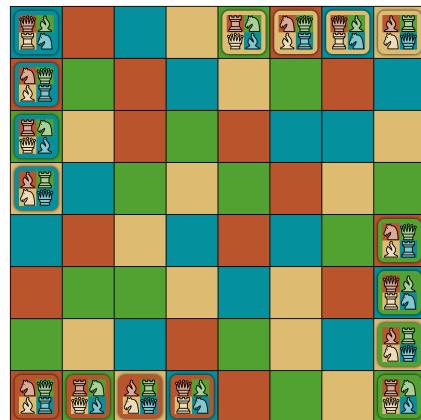


Abbildung 7.4: Das Spielbrett mit den Figuren in Startaufstellung

Jede Figur hat eine Farbe-Rolle-Zuordnung. Heißt, zu jedem der vier verschiedenen Farben auf dem Spielbrett (rot, grün, gelb, blau) wird eine von vier Schach-Rollen (Springer, Dame, Läufer, Turm) zugeordnet. Diese Zuordnung ist fester Bestandteil der Figur und ändert sich im Spielverlauf nicht.

Um nun zu ermitteln, welche Rolle eine Figur zur Zeit inne hat (ergo, wie sie sich bewegen kann), wird das Feld betrachtet, auf der sich die Figur gerade befindet. Anschließend wird der Farbe-Rolle-Zuordnung entnommen, welche Rolle die Figur auf einem Feld der entsprechenden Farbe hat.



Abbildung 7.5: Die Figuren eines Spielers mit den jeweiligen Farbe-Rolle Zuordnungen

Sobald sich eine Figur bewegt hat, befindet sie sich im nächsten Zug auf einem anderen Feld mit einer anderen Farbe. Folglich hat sie nun auch eine andere Rolle, die wieder der Zuordnung zu entnehmen ist.

Spielverlauf

Es ist ein zugbasiertes Spiel. Spieler Rot beginnt. Danach wird im Uhrzeigersinn weitergespielt.

Wer an der Reihe ist, bewegt eine seiner Figuren entsprechend der aktuellen Rolle der Figur. Befindet sich auf dem Zielfeld eine gegnerische Figur, so wird diese geschlagen und ist aus dem Spiel.

Sind alle vier Figuren eines Spielers geschlagen, hat dieser verloren und ist ebenfalls aus dem Spiel.

Es besteht Zugzwang, aber kein Schlagzwang.

Einengung des Spielbretts

Nachdem ein Spieler seine Figur bewegt hat, wird überprüft, ob eine oder mehrere der äußereren Reihen frei geworden sind. D.h. in der entsprechenden Reihe befindet sich keine Spielfigur mehr. Ist dies der Fall, so wird das Spielbrett um diese freien Reihen eingeengt.

Das Spielbrett darf allerdings nicht kleiner als 3x3 Felder werden.

Nach der Einengung ist der nächste Spieler an der Reihe.

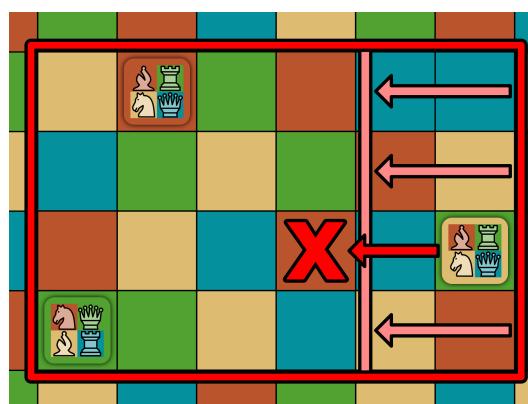


Abbildung 7.6: Einengung des Spielbretts

Spielende

Gewinner ist der Spieler, welcher als einziger noch eine oder mehrere Figuren auf dem Spielbrett übrig hat.

Züge der Spielfiguren

Je nachdem welche Farbe das Feld hat, auf dem sich eine Figur befindet, hat diese Figur eine andere Rolle. Je nach Rolle darf eine Figur nur bestimmte Bewegungen ausführen.

Springer

Der Springer bewegt sich entweder zwei Felder horizontal plus ein Feld vertikal oder zwei Felder vertikal plus ein Feld horizontal. Anders als die anderen Figuren, wird er durch im Weg stehende Figuren nicht behindert. Er “springt” auf sein Zielfeld.

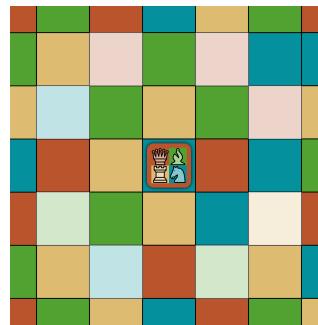


Abbildung 7.7: Züge eines Springers

Läufer

Der Läufer kann sich eine beliebige Anzahl von freien Feldern diagonal von seinem Startfeld aus bewegen. Er kann keine Figuren überspringen, die ihm im Weg stehen.

Steht ihm eine gegnerische Figur im Weg, so kann er diese schlagen. Allerdings endet sein Zug dann auf dem entsprechenden Feld. Eine eigene Figur blockiert sein Fortkommen und kann natürlich nicht geschlagen werden.

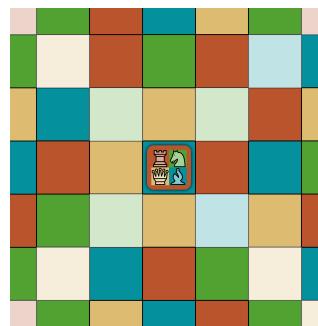


Abbildung 7.8: Züge eines Läufers

Turm

Der Turm kann sich eine beliebige Anzahl von freien Feldern horizontal bzw. vertikal von seinem Startfeld aus bewegen. Er kann keine Figuren überspringen, die ihm im Weg stehen.

Steht ihm eine gegnerische Figur im Weg, so kann er diese schlagen. Allerdings endet sein Zug dann auf dem entsprechenden Feld. Eine eigene Figur blockiert sein Fortkommen und kann natürlich nicht geschlagen werden.

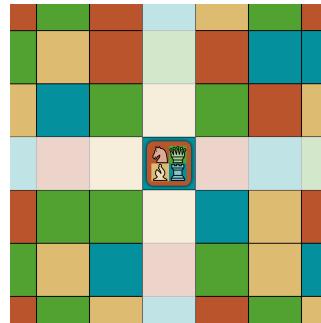


Abbildung 7.9: Züge eines Turms

Dame

Die Dame ist die stärkste Figur im Spiel. Sie kombiniert die beiden Rollen Läufer und Turm in sich. Eine Dame kann sich also beliebig viele freie Felder horizontal, vertikal oder diagonal von ihrem Startfeld aus bewegen. Gegnerische Figuren können geschlagen werden, stoppen aber die Fortbewegung. Eigene Figuren blockieren das weitere Fortkommen.

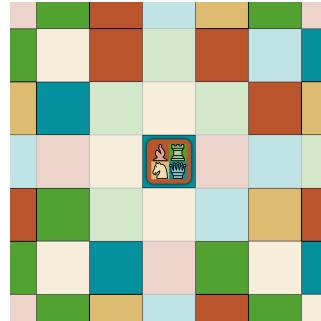


Abbildung 7.10: Züge einer Dame

Spezialfall

Gegen Ende des Spieles, wenn das Brett auf 3x3 Felder geschrumpft ist, kann es zu einem Spezialfall kommen. Unter Umständen kommt eine Figur genau auf dem mittleren Feld zum Stehen und hat nun die Rolle "Springer". In diesem Fall kann sich die Figur nicht mehr bewegen, da die möglichen Springerzüge alle außerhalb des geschrumpften Bretts liegen. Daher wird diese Figur nun direkt aus dem Spiel entfernt.

Ist diese Figur allerdings die allerletzte Figur, die überhaupt noch auf dem Brett stehen geblieben ist, so gewinnt trotzdem der Spieler, dem diese Figur gehört.

Source Code

Der komplette Source Code befindet sich in einem Git-Repository, welches dieser Arbeit auf einer CD beiliegt. Darüber hinaus ist das Projekt auch auf GitHub verfügbar unter: <https://github.com/hd-code/chameleon-chess-ai>.

In dem Repository gibt es eine Datei namens `README.md`. Darin wird die Installation und Verwendung der Software sowie die Projektstruktur erklärt. Alle Berechnungen und Analysen, im Speziellen alle Ergebnisse in den Tabellen im Kapitel Auswertung, können mithilfe des Git-Repositories erneut abgerufen bzw. generiert werden.

Selbstständigkeitserklärung

Ich, Hannes Dröse, versichere hiermit, dass ich die vorliegende Bachelorarbeit mit dem Thema

Implementierung eines Computergegners für Chamäleon Schach nach heuristischem Ansatz

selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Erfurt, 31.08.2020

Hannes Dröse