Vergleich verschiedener Algorithmen für ein N-Spieler schachähnliches Brettspiel

Hannes Dröse[1]

Abstract: Die Implementierung von Computergegnern für N-Spieler-Spiele (mit mehr als zwei Spielern) stellt eine besondere Herausforderung dar. Für die Lösung gibt es eine Vielzahl verschiedener Algorithmen und Varianten. Daher werden in diesem Paper ein paar typische Algorithmen mit einander verglichen. Dabei handelt es sich um den Max^N (mit verschiedenen Pruning Strategien), den paranoiden Minimax und den Hypermax-Algorithmus. Alle Algorithmen werden implementiert und ihre Effektivität wird anhand des Brettspiels "Chamäleon Schach" auf die Probe gestellt.

Keywords: AI; Computergegner; Brettspiel; N-Spieler; MaxN; Paranoid-Minimax; Hypermax; Pruning

1 Einleitung

1982 hat Wolfgang Großkopf, der Großvater des Autors, ein Spiel namens "Chamäleon" entwickelt. Es handelt sich dabei um eine interessante Schach-Variante. Das Spiel ist mehrmals verlegt worden, u.a. 1990 und 1992 vom VSK Verlag. 1991 belegte es sogar den siebten Platz im bekannten Wettbewerb "Spiel des Jahres".

Nun ist eine Neuauflage in digitaler Form unter dem Namen "Chamäleon Schach" geplant, die ebenfalls einen möglichst starken Computergegner enthalten soll. Das Spiel kann nicht nur zu zweit, sondern auch zu dritt oder zu viert gespielt werden. Dadurch entstehen besondere Herausforderungen für die Implementierung des Computergegners. Deswegen werden in diesem Paper verschiedene Algorithmen zur Lösung dieser Aufgabe präsentiert, analysiert und miteinander vergleichen.

2 "Chamäleon Schach"

2.1 Spielprinzip

Zum besseren Verständnis wird hier kurz das Spielprinzip von "Chamäleon Schach" erklärt.

[1] Fachhochschule Erfurt, Altonaer Str. 25, 99085 Erfurt, Deutschland hannes.droese@fh-erfurt.de

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.

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. So ist eine Figur auf einem roten Feld vielleicht ein Turm, auf einem grünen Feld aber ein Springer. Es kommen vier verschiedene Schach-Rollen vor: Springer, Königin, Läufer und Turm. Die Zuordnung von Feldfarbe zu Rolle ist je nach Figur unterschiedlich.

Jeder Spieler startet mit vier Figuren. Wer am Zug ist, wählt eine seiner Figuren aus und bewegt sie entsprechend der Rolle (z.B. ein Turm kann nur horizontal oder vertikal ziehen). Wird ein Feld erreicht, auf dem eine gegnerische Figur steht, kann diese geschlagen werden. Es besteht aber kein Schlagzwang. Danach ist der nächste Spieler an der Reihe.

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

2.2 Spielanalyse

“Chamäleon Schach” ist ein Spiel mit perfekter Information, da es keine Elemente gibt, die nicht allen Spielern bekannt sind. Alle Informationen zum Spiel können mit einem Blick auf das Spielbrett gewonnen werden. [vgl. LR57, s. 58f]

Es gibt immer einen klaren Gewinner, daher machen Kooperationen wenig Sinn. Das Spiel ist strikt kompetitiv und damit ein Nullsummenspiel bzw. allgemeiner gefasst: ein Spiel mit konstanter Summe. [vgl. LR57, s. 164]

Nach der bisherigen Analyse wirkt dieses Brettspiel dem klassischen Schach sehr ähnlich. Für Spiele dieser Art gibt es eine klare Strategie zur Erstellung eines Computergegners: der Minimax-Algorithmus mit Alpha-Beta-Pruning. Die Tatsache, dass “Chamäleon Schach” aber nicht nur zu zweit, sondern auch zu dritt oder zu viert gespielt werden kann, verändert die Sachlage. Der Minmax-Algorithmus funktioniert nur bei Zwei-Spieler-Spielen. Für Mehrspieler-Spiele gibt es verschiedene Ansätze und keinen klar überlegenen Algorithmus, sodass in so einem Fall immer abgewogen werden muss, welcher Algorithmus infrage kommt. Ein direkter Vergleich der Algorithmen gegeneinander ist also nötig. [vgl. St03, s. 2f]

3 Algorithmen

3.1 Grundprinzip

Das Grundprinzip für Computergegner in Brettspielen ist eigentlich immer das gleiche. Es gibt eine heuristische Bewertungsfunktion, mittels der Spielzustände bewertet werden

können. Der Computer generiert also, von der aktuellen Situation ausgehend, die möglichen Züge und das aus dem jeweiligen Zug resultierende Spielbrett. Nun werden diese Spielzustände mit der Bewertungsfunktion bewertet und der Computer wählt den besten Zug aus und führt ihn durch. [vgl. St03, s. 26]

Diese heuristischen Bewertungsfunktionen sind immer sehr individuell vom Spiel abhängig. Sie versuchen, möglichst präzise festzustellen, welcher Spieler gerade die Oberhand hat. [vgl. St03, s. 27] Für dieses Paper nehmen wir die Bewertungsfunktion als gegeben an. Es sollen nur die darauf aufbauenden Algorithmen miteinander verglichen werden.

Die Effektivität der Bewertung kann gesteigert werden, indem nicht nur ein, sondern mehrere Züge in die Zukunft geschaut wird. Das heißt, zu jedem möglichen Zug werden wiederum die darauf möglichen Züge generiert. Es entsteht eine Baumstruktur mit der aktuellen Spielsituation als Wurzel. Bewertet werden nur die Blatt-Knoten des Baumes. Anschließend werden diese Bewertungen mittels einer Entscheidungsregel (die vom konkreten Algorithmus abhängig ist) den Baum aufwärts propagiert. Die Bewertungen auf der ersten Ebene (das sind die möglichen Züge, die der Computer ausführen kann) werden dadurch wesentlich besser und "vorausschauender" bewertet. [vgl. St03, s. 26f]

Die Tiefe des Baums, also wieviele Züge in die Zukunft geblickt wird, hängt von der verfügbaren Rechenzeit ab. Dabei ist es üblich, den Spielbaum iterativ mehrmals hintereinander zu generieren. Jedes mal mit einer größeren Tiefe. Die dabei gefundenen Bewertungen werden jeweils aktualisiert. Das geschieht so lange, wie noch Rechenzeit zur Verfügung steht. Dieser Prozess wird als Iterative Deepening bezeichnet. Die Bewertungen werden umso intelligenter, je tiefer der Baum aufgebaut wird. Daher lässt sich mittels der verfügbaren Rechenzeit auch das Schwierigkeits- bzw. Intelligenzlevel der Algorithmen regeln. [vgl. St03, s. 91f]

3.2 Minimax mit Alpha-Beta-Pruning

In Zwei-Spieler-Spielen ist dies der effektivste Algorithmus und kann als Standardlösung angesehen werden [vgl. St03, s. 3f]. Eine gute Beschreibung und Herleitung geben Knuth und Moore [KM75].

Einer der Spieler wird als Max, der andere als Min bezeichnet. Die Bewertungsfunktion liefert ein skalares Ergebnis. Je höher die Bewertung, desto besser für Spieler Max, je niedriger, desto besser für Spieler Min. Da wir von strikt kompetitiven Spielen sprechen, in denen Koalitionen keinem etwas nützen, ist diese Betrachtung zulässig. Bei der Auswahl des besten Zuges wählt Spieler Max daher stets den am höchsten und Spieler Min den am niedrigsten bewerteten Zug.

Was diesen Algorithmus so stark macht, ist, dass eine sehr effektive Form des Prunings (dt. Beschneiden) auf den Spielbaum angewendet werden kann: Das Alpha-Beta-Pruning.

Im Worst Case hat der Algorithmus eine Laufzeit von $O(b^d)$ (das gilt für alle hier betrachteten Algorithmen). Wobei b der branching factor, also die Anzahl möglicher Züge, die ein Spieler ausführen kann, ist. d ist die Tiefe (engl. depth), also wieviele Züge in die Zukunft geschaut wird. Tatsächlich muss aber nicht in allen Fällen der komplette Spielbaum aufgebaut, durchsucht und bewertet werden. Mit Pruning-Verfahren werden Zweige des Baumes, bei denen bereits absehbar ist, dass sie keine bessere Bewertung liefern werden, beschnitten.

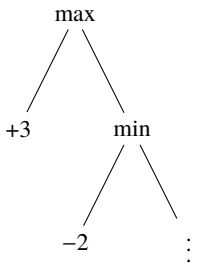

Abb. 1: Alpha-Beta-Pruning. Bereits wenn Min die -2 entdeckt, ist klar, dass Max sich niemals für den rechten Zweig entscheiden wird.

Betrachten wir dazu Abbildung 1. Die Analyse des Spielbaums erfolgt immer über eine Tiefensuche. Nehmen wir an, Spieler Max hat auf dem zuerst bewerteten Zweig (links) eine +3 als Bewertung erhalten. Spieler Max würde sich also für diesen Zweig entscheiden, wenn keine höhere Bewertung gefunden wird. Nehmen wir nun an, dass auf dem nächsten Zweig Spieler Min (ein Zug weiter in der Zukunft) eine -2 als Bewertung erhalten hat. Da Spieler Min immer die kleinstmögliche Bewertung bevorzugt, wird er auf jeden Fall den Zweig mit der -2 oder sogar eine noch kleinere Bewertung wählen. Da Spieler Max aber eine Ebene darüber steht und auf dem linken Zweig bereits ein +3 bekommen kann, ist klar, dass Spieler Max niemals den rechten Zweig wählen wird, sondern stets den linken. Wir können also den kompletten rechten Zweig ignorieren, sobald wir die -2 gefunden haben. Das ist das Alpha-Beta-Pruning.

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.

Durch dieses Verfahren wird eine Laufzeit von $O(b^{\frac{d}{2}})$ im Best Case und $O(b^{\frac{3d}{4}})$ im Average Case erreicht. [vgl. St03, s. 43; zitiert nach Pe84]

Leider reicht ab drei Spielern keine skalare Bewertung der Spielzustände mehr aus. Damit funktioniert dieser Algorithmus für Spiele ab drei Spielern nicht mehr. Sturtevant und

Korf [SK00] geben allerdings eine Lösung dazu: nämlich den paranoiden Ansatz. Wir nehmen also an, dass der Spieler, der am Zug ist, Max ist. Alle seine Gegner bilden eine Koalition gegen ihn und sind Spieler Min. Nun kann wieder der Minimax-Algorithmus mit Alpha-Beta-Pruning verwendet werden.

Sie 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.

In Sturtevants Versuchen u.a. mit dem Spiel Chinese Checkers [St03, s. 118] ist aber der paranoide Ansatz den anderen Algorithmen häufig überlegen gewesen. Unter anderem weil durch die höhere Suchtiefe die Bewertung der Züge effektiver und "weitsichtiger" ist.

3.3 Max^N mit Immediate und Shallow Pruning

Der Max^N-Algorithmus ist von Luckhardt und Irani [LI86] vorgestellt worden. Der Minimax-Algorithmus wird dabei auf N-Spieler erweitert. Die Bewertungsfunktion liefert einen Ergebnisvektor mit je einem Eintrag pro Spieler. Die Spieler wählen nun immer jeweils den Zug mit der höchsten Bewertung an ihrer Stelle im Vektor. Es herrscht aber kein direkter Zusammenhang zwischen den Werten im Vektor vor. Dadurch ist das Pruning in der bisherigen Form nicht mehr möglich.

Es gibt dennoch einige Möglichkeiten des Prunings im Max^N. Keine ist so effektiv wie das Alpha-Beta-Pruning und alle sind an spezielle Bedingungen geknüpft. [vgl. SK00]

3.3.1 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. SK00]

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

3.3.2 Shallow Pruning

Auch diese Art des Prunings ist von Sturtevant und Korf [SK00] beschrieben worden. Die Bedingung ist hier, dass es eine maximale Summe der Elemente im Ergebnisvektor gibt.

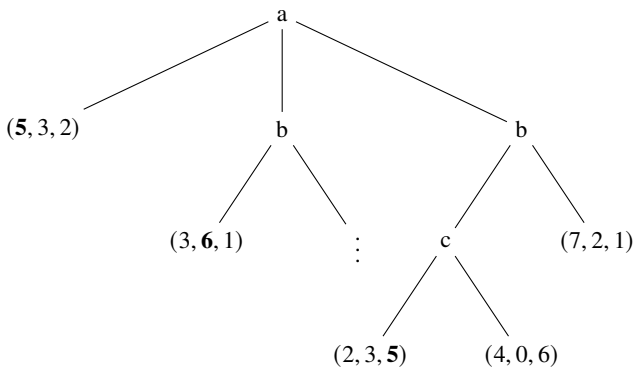

Abb. 2: Shallow Pruning. Die $max\Sigma = 10$ daher wird beim ersten Blatt unter b klar, dass a sich stets für den linken Zweig entscheiden wird.

Betrachten wir dazu Abbildung 2. Spieler a ist am Zug und der erste Wert im Ergebnisvektor gehört zu a. Im linken Zweig kann also eine +5 erreicht werden. Die Summe des Ergebnisvektors ist stets 10. In dem Moment, wo Spieler b im mittleren Zweig die Bewertung $(3, 6, 1)$ erhält, ist klar, dass Spieler a in diesem Zweig maximal noch eine +4 erreichen kann. 6 von 10 Punkten sind ja schon an Spieler b gegangen und dieser wird das Ergebnis für sich eventuell sogar noch weiter maximieren. Daher wird sich Spieler a stets für den linken, anstatt für den mittleren Zweig entscheiden und der mittlere Zweig muss nicht weiter betrachtet werden.

Diese Art des Prunings kann nur zwischen Vater- und Kindknoten durchgeführt werden. Wenn Spieler c im rechten Zweig die Bewertung $(4, 0, 6)$ findet, könnte man ja vermuten, dass der komplette Zweig ignoriert werden kann. Ähnlich wie es im mittleren Zweig der Fall gewesen ist. Auf den zweiten Blick sehen wir aber, dass sich Spieler b im normalen Max^N ohne Pruning für den Zweig ganz rechts entscheiden würde. Dadurch bekommt Spieler a eine Bewertung von $(7, 2, 1)$ und wird sich für den rechten Zweig entscheiden. Durch das Pruning zwischen Großvater- und Enkelknoten, hätten wir diese Bewertung allerdings nicht gefunden.

Dadurch beschränkt sich das Pruning nur auf unmittelbar verwandte Knoten und die Effektivität sinkt. Im Best Case tendiert diese Lösung gegen $O(b^{\frac{d}{2}})$. [vgl. St03, s. 52f]

3.3.3 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 Vorrausetzungen in jeder Implementierung des Max^N-Algorithmus geschaffen werden.

Die Idee dazu hat bereits Sturtevant [St03, s. 73] gehabt. Dazu ist folgende Modifikation notwendig:

Angenommen unsere Bewertungsfunktion liefert uns einen Bewertungsvektor $\vec{v} \in \mathbb{R}_0^{+N}$. Dann kann dieser zu einem Vektor $\vec{v}^*$ umgewandelt werden, der die Bedingungen erfüllt. Dazu müssen lediglich die Elemente von $\vec{v}$ jeweils durch die Summe von $\vec{v}$ geteilt werden. Also: $v_i^* = \frac{v_i}{\Sigma \vec{v}}$.

Dadurch ist die Summe von $\vec{v}^*$ stets 1. Der höchstmögliche Wert, den ein Element in $\vec{v}^*$ erreichen kann, ist ebenfalls 1. Damit sind beide Vorraussetzungen für Immediate und Shallow Pruning gegeben.

Anmerkung: $\vec{v}$ darf kein Nullvektor sein. Außerdem: wenn Elemente in $\vec{v}$ auch negative Werte annehmen können, dann sollte zunächst zu allen Elementen der Betrag des kleinsten Elements addiert werden. Dadurch gibt es keine negativen Werte im Vektor mehr. Das Verhältnis der Elemente zueinander wird dadurch allerdings minimal verzerrt. Die meisten Bewertungsfunktionen liefern aber ohnehin nur positive Bewertungen.

3.4 Hypermax

Der Hypermax-Algorithmus ist von Mikael Fridenfalk [Fr14] 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 unserer 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 in $\vec{v}$ abgezogen wird. Also: $v_i^* = v_i - avg(\vec{v})$.

Der Algorithmus funktioniert nun so, dass in einem Vektor (den Fridenfalk konsequent $\vec{\alpha}$ nennt) die jeweils bisher besten Ergebnisse für die jeweiligen Spieler gespeichert werden. Wann immer eine bessere Bewertung gefunden wird, wird das zum 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.

Er gibt aber auch zu bedenken, dass durch die Transformation in den Zero-Space die Verhältnisse der Bewertungen zueinander geringfügig verzerrt werden. Außerdem können durch das Pruning Zweige beschnitten werden, obwohl sie eigentlich auf einer tieferen Ebene eine bessere Bewertung liefern würden. Das Problem scheint ähnlich gelagert zu sein, wie der Grund, warum das Shallow Pruning im Max^N nur zwischen Vater und Kindknoten möglich ist, ohne das Ergebnis zu verändern. In dieser Richtung wären weitere Analysen nötig.

Dennoch sagt Fridenfalk, dass der Hypermax eine gute Approximation des ursprünglichen Max^N liefern und durch das effektivere Pruning weiter in die Zukunft blicken kann. Die Argumentation ist also ähnlich zum paranoiden Ansatz. Nicht die Genauigkeit der Berechnung, sondern die höhere Suchtiefe sollen den entscheidenden Vorteil liefern.

Betrachten wir nun, wie sich die Algorithmen gegeneinander behaupten können.

4 Vergleich der Algorithmen

4.1 Vorbereitung

Für den Vergleich sind die folgenden Algorithmen implementiert worden:

- Max^N (ohne Pruning)
- Max^NIS (Max^N mit Immediate und Shallow Pruning)
- Paranoid (paranoider Minimax mit Alpha-Beta-Pruning)
- Hypermax

“Chamäleon Schach” kann zu viert gespielt werden, daher treten alle vier Algorithmen gleichzeitig gegeneinander an. Es macht durchaus einen Unterschied, welcher Spieler der Startspieler ist bzw. welcher als zweites, drittes oder viertes zieht. Daher sind insgesamt 24 Spiele gespielt worden, in allen möglichen Anordnung der vier Spieler um das Spielbrett (4! = 24 verschiedene Anordnungen).

Die Algorithmen verwenden das Iterative Deepening und haben eine Sekunde Zeit, den optimalen Zug zu berechnen. In der App später werden sie ebenfalls eine Sekunde Zeit haben. Die Algorithmen sind in Node.js implementiert worden und der Test ist auf einem MacBook Pro 13"von 2018 mit einem Intel Core i5 Prozessor und 16GB RAM durchgeführt worden.

In den Originalregeln ist kein Unentschieden vorgesehen gewesen. Im Spiel mit realen Menschen ist es bisher auch noch nie zu einem Unentschieden gekommen. Bei den

Algorithmen kann es aber vorkommen, dass sie gegen Ende des Spieles einen zyklischen Ablauf finden. Sie “tanzen” dann mehr oder weniger die ganze Zeit umeinander, ohne dass einer den anderen schlagen und das Spiel entscheiden könnte. Daher sind die Spiele auf 100 Züge begrenzt worden (ein typisches Spiel dauert zwischen 40 und 60 Zügen). Wenn der 100. Zug erreicht ist, wird ein Unentschieden für alle noch lebenden Spieler erklärt.

4.2 Auswertung

Algorithmus	wins	draws	losses	depthAvg	depthMedian
Max^N	8	0	16	5.02	4.08
Max^NIS	10	1	13	7.38	4.08
Hypermax	2	0	22	5.49	4.04
Paranoid	3	1	20	6.39	4.2

Tab. 1: Ergebnisse der 24 Spiele der verschiedenen Algorithmen gegeneinander.

Die Tabelle zeigt die Ergebnisse der 24 Spiele. Es wird zu jedem Algorithmus angegeben, wie häufig dieser gewonnen, unentschieden gespielt oder verloren hat. Bei jeder Berechnung ist die vom Algorithmus erreichte Suchtiefe gespeichert worden. Die Berechnung wird sofort abgebrochen, wenn die vorgegebene Zeit von einer Sekunde erreicht ist. Daher wird die Suchtiefe mit Kommastellen angegeben (3.5 würde bedeuten, dass die Hälfte der Züge mit einer Suchtiefe von 3 und die andere Hälfte mit 4 bewertet worden ist). Die Spalte “depthAvg” enthält den Durchschnitt aller erreichten Suchtiefen je Algorithmus; “depthMedian” den Median derselbigen.

Zunächst fällt auf, dass die durchschnittliche Suchtiefe stark von ihrem Median abweicht. Das liegt daran, dass gegen Ende des Spieles, wenn nur noch sehr wenige Figuren übrig sind, sehr viel tiefer gesucht werden kann. Algorithmen, die sehr häufig gewonnen haben, werden dadurch wesentlich mehr hohe Ausreißer in der Suchtiefe haben. Daher ist der Median besser zum Vergleichen geeignet.

Der Paranoid hat im Median die höchste Suchtiefe erreicht. Dies ist nicht verwunderlich, da das Alpha-Beta-Pruning ja das effektivste Pruning-Verfahren darstellt. Der Hypermax hat tatsächlich die geringste Suchtiefe erreicht. Dies lässt sich dadurch erklären, dass der Hypermax fast alle Spiele verloren hat und daher fast ausschließlich Werte aus der rechenaufwendigen Anfangsphase des Spieles vorliegen. Wahrscheinlich ist es der recht hohen Effizienz des Prunings vom Hypermax zuzuschreiben, dass seine Ergebnisse überhaupt in der Nähe der anderen Algorithmen liegen.

Die Max^N-Algorithmen haben im Median die exakt gleiche Suchtiefe erreicht. In Bezug auf den Durchschnitt liegt der Max^{-N}IS allerdings deutlich höher. Das liegt daran, dass Immediate

und Shallow Pruning erst gegen Ende des Spieles ihre volle Effektivität entwickeln. Dadurch werden besonders am Ende eines Spieles extrem hohe Suchtiefen erreicht. Im Hauptteil des Spielverlaufes hat das Pruning hingegen kaum einen Effekt.

Der Hypermax hat fast jedes Spiel verloren. Fridenfalk führt an, dass der Hypermax nur eine Approximation des Max^N ist und die konkreten Bewertungen der beiden Algorithmen durchaus unterschiedlich ausfallen können. Diese Unterschiede sind für "Chamäleon Schach" so signifikant, dass der Hypermax keine Chance gegen die anderen Algorithmen hat.

Auch der Paranoid hat, trotz höherer Suchtiefe, fast alle Spiele verloren. Sturtevant hat postuliert, dass die paranoide Betrachtung zu einer fehlerhaften Spieleweise führen kann, da die angenommen Koalition zwischen den Gegnern gar nicht existiert. Dieser Fehler potenziert sich außerdem mit steigender Suchtiefe. [vgl. St03, s. 133] Es ist also gut möglich, dass der Paranoid bei geringen Suchtiefen (oder mit weniger Rechenzeit) besser abschneidet. In diesem Versuch hat der Paranoid allerdings keine Chance gegen die Max^N-Algorithmen gehabt.

Die Max^N-Algorithmen schneiden in diesem Versuch am besten ab, mit deutlichem Vorsprung. Der Max^NIS ist seinem Bruder etwas überlegen. Im Hauptteil des Spieles hat das Pruning, wie erwähnt, kaum einen Effekt. Gegen Ende des Spieles erreicht der Max^NIS aber höhere Suchtiefen und spielt damit tatsächlich etwas besser als der Max^N.

5 Fazit

In diesem Paper sind verschiedene Algorithmen für strikt kompetitive N-Spieler-Brettspiele mit perfekter Information miteinander verglichen worden.

Hypermax und Paranoid, welche die beiden Algorithmen mit dem effektivsten Pruning-Verfahren sind, haben in diesem Versuch am schlechtesten abgeschnitten. Der Hypermax hat außerdem nur eine mittelmäßige Suchtiefe erreicht, was allerdings an der Menge der verlorenen Spiele liegen kann. Beide Algorithmen liefern keine optimale Bewertung ab, sollen aber durch besseres Pruning und höhere Suchtiefe diesen Nachteil ausgleichen. Die erreichten Suchtiefen haben bei allen Algorithmen sehr nah beieinander gelegen. In Zukunft könnte untersucht werden, wie sich verschiedene Rechenzeiten auf die Suchtiefe und die Effektivität der Spielweise auswirken. Insgesamt scheint die besondere Beschaffenheit von Mehrspieler-Spielen aber dafür zu sorgen, dass ein effizienteres Pruning nicht mit einer effektiveren Spielweise gleichzusetzen ist.

Die klaren Gewinner sind die "klassischen" Max^N-Algorithmen gewesen. Mit Immediate und Shallow Pruning kann die Suchtiefe und die Effektivität der Bewertungen vor allem gegen Ende des Spieles gesteigert werden. Für "Chamäleon Schach" ist damit der "beste" Algorithmus für den Computergegner gefunden: der Max^N mit Immediate und Shallow Pruning.

In Sturtevants Versuchen u.a. mit Chinese Checkers ist der paranoide Ansatz meist der stärkste gewesen. Für “Chamäleon Schach” hat sich diese Erkenntnis nicht bestätigt. N-Spieler-Spiele bleiben damit ein schwieriges Problem, zu dem es keine Standardlösung gibt. Stattdessen müssen die verschiedenen Algorithmen jeweils mit dem entsprechenden Spiel getestet und verglichen werden.

Literatur

[Fr14] Fridenfalk, M.: N-Person Minimax and Alpha-Beta Pruning. In: NICOGRAPH International 2014. S. 43–52, 2014.

[KM75] Knuth, D. E.; Moore, R. W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6/4, S. 293–326, 1975.

[LI86] Luckhart, C. A.; Irani, K. B.: An Algorithmic Solution of N-Person Games. In: AAAI. Bd. 86, S. 158–162, 1986.

[LR57] Luce, R. D.; Raiffa, H.: Games and Decisions: Introduction and Critical Survey. John Wiley, New York, 1957.

[Pe84] Pearl, J.: Heuristics Addison-Wesley. Reading, MA/, 1984.

[SK00] Sturtevant, N. R.; Korf, R. E.: On Pruning Techniques for Multi-Player Games. AAAI-2000 49/, S. 201–207, 2000.

[St03] Sturtevant, N. R.: Multi-Player Games: Algorithms and Approaches, Diss., University of California, 2003.