

Othello

— Entwicklung einer KI für das Spiel —

Patrick Müller, Max Zepnik

20. Januar 2019

Inhaltsverzeichnis


Kapitel 1


Einleitung

Computergegner ..

. test..

text1 ...

am Ende schrei-
ben 

auf Fazit bezie-
hen? 

Kapitel 2

Grundlagen

2.1 Spieltheorie

Einleitung fehlt

Definition 1 (Spiel (Game)) Ein [Game](#) besteht aus einem Tupel der Form

$$\mathcal{G} = \langle S_0, \text{player}, \text{actions}, \text{result}, \text{terminalTest}, \text{utility} \rangle$$

S_0 beschreibt den Startzustand des Spiels.

PLAYER ist auf der Menge der Spieler definiert und gibt den aktuellen Spieler zurück.

ACTIONS gibt die validen Folgezustände eines gegebenen Zustands zurück.

RESULT definiert das Resultat einer durchgeführten Aktion a und in einem Zustand s .

TERMINALTEST prüft ob ein Zustand s ein Terminalzustand, also Endzustand, darstellt.

UTILITY gibt einen Zahlenwert aus den Eingabewerten s (Terminalzustand) und p (Spieler) zurück.

Positive Werte stellen einen Gewinn, negative Werte einen Verlust dar.

quelle S.162

Definition
Statesrdavor

Eine spezielle Art von Spielen sind [Nullsummenspiele](#).

Definition 2 (Nullsummenspiele) In einem [Nullsummenspiel](#) ist die Summe der utility Funktion eines Zustands über alle Spieler 0. Dies bedeutet, dass wenn ein Spieler gewinnt mindestens ein Gegenspieler verliert.

Durch den Startzustand S_0 und der Funktion ACTION wird ein [Spielbaum \(Game Tree\)](#) aufgespannt.

Definition 3 (Spielbaum (Game Tree)) Ein [Spielbaum](#) besteht aus einer einzigen Wurzel, welche einen bestimmten Zustand (meistens S_0) darstellt. Die Kindknoten der Wurzel stellen die durch ACTIONS erzeugten Zustände dar. Die Kanten zwischen der Wurzel und den Kindknoten stellen jeweils die durchgeführte Aktion dar, die ausgeführt wurde um vom State s zum Kindknoten zu gelangen.

Definition 4 (Suchbaum (Search Tree)) Ein [Suchbaum](#) ist ein Teil des Spielbaums.

[?]

Überleitung
einfügen

2.2 Spielstrategien

Es gibt verschiedene Spielstrategien. Im Folgenden werden diese kurz erläutert und anschließend verglichen.

2.2.1 Min-Max

Der erste hier erläuterte Strategie ist der Min-Max Algorithmus. Dieser ist folgendermaßen definiert:

$$MinMax(s) = \begin{cases} Utility(s); & \text{wenn } TerminalTest(s) == \text{true} \\ \max(\{a_e \in Actions(s) | MinMax(Result(s, a))\}); & \text{wenn Spieler am Zug} \\ \min(\{a_e \in Actions(s) | MinMax(Result(s, a))\}); & \text{wenn Gegner am Zug} \end{cases}$$

Der Spieler sucht den bestmöglichen Zug aus `ACTIONS`, der ihm einen für seine Züge einen Vorteil schafft aber gleichzeitig nur „schlechte“ Zugmöglichkeiten für den Gegner generiert. Der Gegner kann dadurch aus allen ehemals möglichen Zügen nicht den optimalen Zug spielen, da dieser in den aktuell enthaltenen Zügen nicht vorhanden ist. Er wählt aus den verfügbaren `ACTIONS` nach den gleichen Vorgaben seinen besten Zug aus.

Die Strategie ist eine Tiefensuche und erkundet jeden Knoten zuerst bis zu den einzelnen Blättern bevor ein Nachbarknoten ausgewählt wird. Dies setzt das mindestens einmalige Durchlaufen des gesamten Search Trees voraus. Bei einem durchschnittlichen Verzweigungsfaktor von f bei einer Tiefe von d resultiert daraus eine Komplexität von $O(d^f)$. Bei einem einmaligen Erkunden der Knoten können die Werte aus den Blättern rekursiv von den Blättern zu den Knoten aktualisiert werden. Dadurch muss im nächsten Zug nur das Minimum aus `ACTIONS` ermittelt werden, da alle Kindknoten schon evaluiert wurden. Für übliche Spiele kann die Min-Max-Strategie allerdings nicht verwendet werden, da die Komplexität zu hoch für eine akzeptable Antwortzeit ist und der benötigte Speicherplatz für die berechneten Zustände sehr schnell wächst.

2.2.2 Alpha-Beta Pruning

Der Min-Max Algorithmus berechnet nach dem Prinzip „depth-first“ stets den kompletten Game Tree. Bei der Betrachtung des Entscheidungsverhaltens des Algorithmus fällt jedoch schnell auf, dass ein nicht unerheblicher Teil aller möglichen Züge gar nicht erst in Betracht gezogen wird. Dies geschieht aufgrund der Tatsache, dass diese Züge in einem schlechteren Ergebnis resultieren würden als die letztendlich ausgewählten.

Dem Alpha-Beta Pruning Algorithmus liegt der Gedanke zugrunde, dass die Zustände, die in einem realen Spiel nie auftreten würden auch nicht berechnet werden müssen. Damit steht die dafür regulär erforderliche Rechenzeit und der entsprechende Speicher dafür zur Verfügung andere, vielversprechendere Zweige zu verfolgen.

Demonstration an einem Beispiel

Um den Algorithmus zu verdeutlichen betrachten wir das, an [?] angelehnte, folgende Beispiel. Das dargestellte Spiel besteht aus lediglich zwei Zügen, die abwechselnd durch die Spieler gewählt werden. An den Knoten der untersten Ebene des Game Tree werden die Werte der Zustände gemäß der `UTILITY` Funktion angegeben. Die Werte α und β geben den schlecht möglichsten bzw. den bestmöglichen Spielausgang für einen Zweig, immer aus der Sicht des beginnenden Spielers, an. Die ausgegrauten Knoten wurden noch nicht betrachtet.

Betrachten wir nun den linken Baum in der zweiten Zeile: Der Algorithmus beginnt damit alle möglichen Folgezustände bei der Wahl von B als Folgezustand zu evaluieren. Dabei wird zunächst der Knoten E betrachtet und damit der Wert 5 ermittelt. Dies ist der bisher beste Wert. Er wird als β gespeichert. Eine Aussage über den schlechtesten Wert kann noch nicht getroffen werden.

Im nachfolgenden Game Tree wird der nächste Schritt verdeutlicht. Es wird der Knoten F betrachtet. Dieser hat einen Wert von 13. Am Zuge ist jedoch der zweite Spieler. Dieser wird, geht man davon aus, dass er ideal spielt, jedoch keinen Zug wählen der ein besseres Ergebnis für den Gegner bringt als unbedingt nötig. Der bestmögliche Wert für den ersten Spieler bleibt damit 5.

Warum

Nach der Auswertung des Knotens G steht fest, dass es keinen besseren und keinen schlechteren Wert aus Sicht des ersten Spielers gibt. Daraufhin wird die 5 auch als schlechtester Wert in α gespeichert. Ausgehend von A ist der schlechteste Wert damit 5 ggf. kann jedoch noch ein besseres Ergebnis herbeigeführt werden. α wird entsprechend gesetzt und β verbleibt undefiniert.

Nun werden die Kindknoten von C betrachtet. Mit einem Wert von 3 wäre der Knoten H das bisher beste Ergebnis für die Wahl von C. Der Wert wird entsprechend gespeichert. Würde C gewählt gäbe man dem Gegenspieler die Chance ein im Vergleich zu der Wahl des Knotens B schlechteres Ergebnis herbeizuführen. Da Ziel des Spielers jedoch ist die eigenen Punkte zu maximieren gilt es diese Chance gar nicht erst zu gewähren. Entsprechend werden die Auswertung der weiteren Knoten abgebrochen.

Der Kindknoten K des Knotens D ist mit einem Wert von 42 vielversprechend und wird in β gespeichert. Da dieser Wert größer ist als die gespeicherten 5 wird auch der entsprechende Wert von A aktualisiert. Der anschließend ausgewertete Knoten L ermöglicht nun ein schlechteres Ergebnis von 6 β muss also aktualisiert werden. Der Knoten M liefert schließlich den schlechtesten Wert von 1. Da der Gegenspieler im Zweifel diesen Wert wählen würde bleibt der bisher beste Wert das Ergebnis in E. In A wird der Spieler daher B auswählen

Dieses einfache Beispiel zeigt bereits recht gut wie die Auswertung von weiteren Zweigen vermieden werden kann. In der Praktischen Anwendung befinden sich die wegfallenden Zustände häufig nicht nur in den Blättern des Baumes sondern auch auf höheren Ebenen. Der eingesparte Aufwand wird dadurch häufig noch größer.

Implementierung

Nachfolgend wird eine Pseudoimplementierung des um Alpha-Beta Pruning erweiterten MinMax Algorithmus angegeben.:

```
global Suchtiefe
int minMax(Spiel AktuellerZustand, int Spieler, int Tiefe, int alpha, int beta) {
    if (Tiefe == 0) {
        return Utility(AktuellerZustand, Spieler);
    }
    int bisherigerMaximalWert = alpha
    Zuege = mengeDerFolgezuege(aktuellerZustand);
    for (Zug z in Zuege) {
        Spiel NeuerZustand = waehleZug(Aktueller Zustand, z)
        wert = -minMax(NeuerZustand, anderer(Spieler), Tiefe-1, -beta, -bisherigerMaximalWert)
        if (wert > bisherigerMaximalWert) {
            bisherigerMaximalWert = wert
            if (bisherigerMaximalWert >= beta) {
                break;
            }
        }
        if (Tiefe = Suchtiefe) {
            speichereZug(z)
        }
    }
}
return bisherigerMaximalwert;
}
```

Listing 2.1: Pseudoimplementierung Alpha-Beta Pruning

Es handelt sich um eine rekursive Implementierung. Im Basisfall ist der Game Tree bereits bis in die angegebene Suchtiefe erforscht (Zeile 3). In diesem Fall wird der Wert der Utility Funktion für den aktuellen Spieler bei dem aktuellen Zustand zurückgegeben (Zeile 4).

Handelt es sich nicht um einen solchen Fall werden alle möglichen Folgezüge berechnet (Zeile 7) und dann einzeln betrachtet in dem er ausgeführt wird. (Zeile 8f). Zuvor wird dazu jedoch der bisherige Maximalwert gespeichert (Zeile 6). Um den Wert des Zuges zu bestimmen wird rekursiv die minMax-Methode erneut aufgerufen. Dabei wird entsprechend der Neue Zustand, der andere Spieler und eine um die um eins verringerte Tiefe übergeben. Der beste Wert für den anderen Spieler ist der schlechteste Wert für den ersten Spieler. Daher wird der bisherige Wert von beta als alpha übergeben. Der bisher beste Wert ist aus Sicht des anderen Spielers der schlechteste daher wird dieser als neues beta Übergeben. Da die Utility Funktion so implementiert ist, dass die Summe der Wertigkeiten eines Zustandes Null ergibt muss noch das Vorzeichen geändert werden (Zeile 9). Ist der neue Wert größer als der bisherige Maximalwert (Zeile 11) so wird dieser aktualisiert (Zeile 12). Da der zweite Spieler versucht die Punktzahl des Gegners zu maximieren bricht dieser die Auswertung aller Zweige ab, bei denen ein Ergebnis, welches besser ist als das bisher schlechteste Ergebnis, möglich wird (Zeile 13f). Abschließend wird der ausgewertete Zug gespeichert um ihn später ausführen zu können (Zeile 16).

Ordnung der Züge

Wie in obigen Beispiel an den Zweigen unter dem Knoten C zu sehen war kann, je nach der Reihenfolge in der die Folgezüge untersucht werden, die Auswertung eines Folgezustandes früher oder später abgebrochen werden. Optimalerweise werden die besten Züge, also jene Züge die einen möglichst frühen Abbruch der Betrachtung eines Knotens herbeiführen zuerst betrachtet. Um dies Abschätzen zu können bedient man sich in der Praxis einer Heuristik die Aussagen über die Güte eines Zuges im Vergleich zu den übrigen Zügen zulässt. Anhand dieser Heuristik kann dann die Reihenfolge der Auswertung einzelner Folgezustände dynamisch angepasst werden.

2.2.3 Suboptimale Echtzeitentscheidungen

Selbst die gezeigten Verbesserung des MinMax-Algorithmus besitzt noch einen wesentlichen Nachteil. Da es sich um einen "depth-first" Algorithmus handelt muss jeder Pfad bis zu einem Endzustand betrachtet werden um eine Aussage über den Wert des Zuges treffen zu können. Dem steht jedoch die Tatsache entgegen, dass in der Praxis eine Entscheidung möglichst schnell, idealer Weise innerhalb weniger Minuten, getroffen werden soll. Hinzu kommt, dass je nach der verwendeten Datenstruktur für ein Spiel bei entsprechend hohem Verzweigungsfaktor und einer großen Anzahl von Zügen der Hauptspeicher eines handelsüblichen Computers nicht mehr ausreicht um diese zu fassen

Es gilt also eine Möglichkeit zu finden, die Auswertung des kompletten Baumes zu vermeiden.

Heuristiken

Dieses Problem lösen sogenannte Heuristiken. Dabei handelt es sich um eine Funktion die den Wert eines Spielzustandes annähert.

Die Nutzung der Heuristik wird vereinfacht, wenn Sie so definiert ist, dass sie, sofern es sich um einen Endzustand handelt den Wert der Utility Funktion zurückgibt. Der Vorteil dieses Verhaltens wird im nächsten Abschnitt betrachtet.

Kommt eine Heuristik zur Anwendung so ist die Genauigkeit mit der diese den tatsächlichen Wert approximiert der wesentliche Aspekt der die Qualität des Spiel-Algorithmus ausmacht. Um zu verhindern, das versehentlich

consistent/admiss

die besten Züge nicht betrachtet werden, ist es essentiell, dass eine Heuristik den tatsächlichen Wert eines Zustandes nie überschätzt. Das unterschätzen des Wertes hingegen ist möglich darf im Sinne der Genauigkeit der Heuristik jedoch nicht allzu ungleichmäßig Auftreten.

Abschnittskriterium der Suche

Gibt die Heuristik im Falle eines Endzustandes den Wert der Utility Funktion zurück, so kann die oben gezeigte Implementierung so angepasst werden, dass statt der Utility Funktion einfach die Heuristik ausgewertet wird. Dadurch muss nicht mehr der Vollständige Zweig durchsucht werden und das Abbrechen nach einer gewissen Suchtiefe wird möglich.

Forward pruning

Forward pruning durchsucht nicht den kompletten Game Tree, sondern durchsucht nur einen Teil. Eine Möglichkeit ist eine Strahlensuche, welche nur die „besten“ Züge durchsucht. Die Züge mit einer geringen Erfolgswahrscheinlichkeit werden abgeschnitten und nicht bis zum Blattknoten evaluiert. Durch die Wahl des jeweils wahrscheinlichsten Zuges können aber auch sehr gute bzw. schlechte Züge nicht berücksichtigt werden, da sie eine geringe Wahrscheinlichkeit besitzen. Durch das Abschneiden von Teilen des Game Tree wird die Suchgeschwindigkeit deutlich erhöht. Der in dem Othello-Programm „Logistello“ verwendete „Probcut“ erzielt außerdem eine Gewinnwahrscheinlichkeit von 64% gegenüber der ursprünglichen Version ohne Forward pruning.

Search versus lookup

Viele Spiele kann man in 3 Haupt-Spielabschnitte einteilen:

- Eröffnungsphase
- Mittelspiel
- Endphase

In der Eröffnungsphase und in der Endphase gibt es im Vergleich zum Mittelspiel wenige Zugmöglichkeiten. Dadurch sinkt der Verzweigungsfaktor und die generelle Anzahl der states. In diesen Phasen können die optimalen Spielzüge einfacher berechnet werden. Eine weitere Möglichkeit besteht aus dem Nachschlagen des Spielzustands aus einer Lookup-Tabelle.

Dies ist sinnvoll, da gewöhnlicherweise sehr viel Literatur über die Spieleröffnung des jeweiligen Spiels existiert. Auch über die Endzustände in der Schlussphase des Spiels findet sich Literatur. Das Mittelspiel jedoch hat zu viele Zugmöglichkeiten, um eine Tabelle der möglichen Spielzüge bis zum Spielende aufstellen zu können. In dem Kapitel ?? werden die bekanntesten Eröffnungsstrategien aufgelistet.

2.3 Monte Carlo Tree Search

Die sogenannte Monte-Carlo Tree Search (MCTS - Monte-Carlo Baumsuche) bedarf im Gegensatz zu den bisher gezeigten Strategien in ihrer Reinform keine Heuristik. Dabei werden zufällige Spiele gespielt. Aus einem einzigen solchen Spiel lässt sich kaum eine Erkenntnis ableiten; aus einer Vielzahl von zufälligen Spielen lässt sich jedoch bei einer ausreichend großen Anzahl die optimale Lösung bestimmen.

2.3.1 Funktionsweise

[?] beschreiben das Verfahren als einen vierstufigen Prozess zum Aufbau eines Game Trees.

Selection

Ist der Ausgangszustand bereits bekannt, also bereits im Game Tree enthalten, so wird der Folgezustand aufgrund der vorhandenen Daten gewählt. In der Regel stehen hier solche Folgezustände zu denen bereits Daten vorhanden sind und solche die bisher unbekannt sind zur Verfügung. Die Schwierigkeit liegt nun darin zu entscheiden, ob jener Folgezustand der am vielversprechendsten ist (exploitation) oder ein bisher unbekannter Zustand der unter Umständen ein besseres Ergebnis liefern könnte (exploration) gewählt wird. Die bei der Auswahl angewandte Strategie wird als Selection Policy bezeichnet. Genauer wird diese Problemstellung im Abschnitt ?? behandelt.

Expansion

Wenn ein Zustand erreicht wird, der bisher nicht im Game Tree enthalten ist, so wird dieser hinzugefügt. Durch die folgenden beiden Schritte werden dann Informationen zu diesem Zustand gespeichert.

Simulation

Nun werden bis zum Erreichen eines Terminalzustandes zufällige Züge durchgeführt. In weiteren Optimierungen kann hier eine Heuristik eingeführt werden um vielversprechende Züge zuerst zu erkunden.

Backpropagation

Im letzten Schritt werden dann die gespeicherten Informationen durch Backpropagation angepasst. Dabei wird die Häufigkeit des Besuchs eines Zustandes, sowie jeweils die Häufigkeit eines Gewinn bzw. Verlusts bei der Wahl dieses Zustandes gespeichert. Der Wert des Zustandes kann nun durch die Anzahl der Gewinne bei Wahl der Aktion durch die Besuchshäufigkeit angenähert werden.

Nach dem derartigen Aufbau des Game Trees wurde aufgrund der Auswahlbedingung im Selection-Schritt jener Folgezustand am häufigsten erkundet der am Erfolgversprechendsten ist. Im tatsächlichen Spiel wird daher der Zug durchgeführt der beim Aufbau des Baumes am häufigsten durchgeführt wurde.

2.3.2 Die Selection Policy

Das Problem des Auswählens des durchzuführenden Zuges ist analog zu dem sogenannten K-Armed Bandit Problem. Dabei spielt der Spieler an einem mehrarmigen Banditen, also einem Glücksspielautomaten. Bei der Wahl eines Armes wird mit einer bestimmten Wahrscheinlichkeit ein Gewinn ausgeschüttet. Damit steht der Spieler vor jedem Zug vor der Wahl: Er kann entweder den Arm betätigen der nach seinem Wissen den höchsten Gewinn verspricht, geht dabei aber das Risiko ein einen Arm der einen bedeutend höheren Gewinn ermöglicht nicht zu betätigen, oder er kann einen Arm zu dem ihm bisher noch keine Informationen vorliegen spielen um ggf. einen Arm mit besseren Chancen zu finden. In der Literatur wird dieses Problem als Exploration-Exploitation Dilemma bezeichnet.

Das Problem kann als eine Reihe von unabhängigen Zufallsvariablen $X_{i,n}$ betrachtet werden. Dabei steht $1 \leq i \leq K$ für den Arm des Banditen und $n \geq 1$ für den Zug. Das Spielen eines Armes i ergeben die Gewinne $X_{i,1}, X_{i,2}, \dots, X_{i,n}$ die gemäß einer zunächst unbekannten Vorschrift mit dem Erwartungswert μ_i berechnet wird.

[?]

Nachfolgend finden sich einige Strategien um die Wahl des Armes bzw. Folgezustandes vorzunehmen:

ϵ -Greedy

Die ϵ -greedy Policy ist eine vergleichsweise einfache Variante zur Lösung des Problems. Um der exploration Rechnung zu tragen wird dabei mit einer festen Wahrscheinlichkeit ϵ ein zufälliger Zug ausgewählt. Andernfalls kommt jener Zug zum Einsatz der den nach aktuellem Wissensstand höchsten Gewinn verspricht. In einer angepassten Variante, vorgeschlagen durch [?], wird die Wahrscheinlichkeit ϵ je nach Wissenstand des Spielers angepasst. Zu Beginn ist sie damit bspw. vglw. hoch während sie bei zunehmender Sicherheit verringert wird.

Regret

Die Regret-Policy versucht den Verlust durch die Wahl eines anderen als den nach derzeitigem Stand optimalen Schrittes so gering wie möglich zu halten. Dieser Verlust wird für n Durchgänge wie folgt berechnet:

$$R_N = \mu^* n - \mu_j \sum_{j=1}^K E[T_j(n)]$$

Dabei steht μ^* für den erwarteten Maximalgewinn und $E[T_j(n)]$ für die erwartete Anzahl der Züge bei denen der Arm j gewählt wurde.

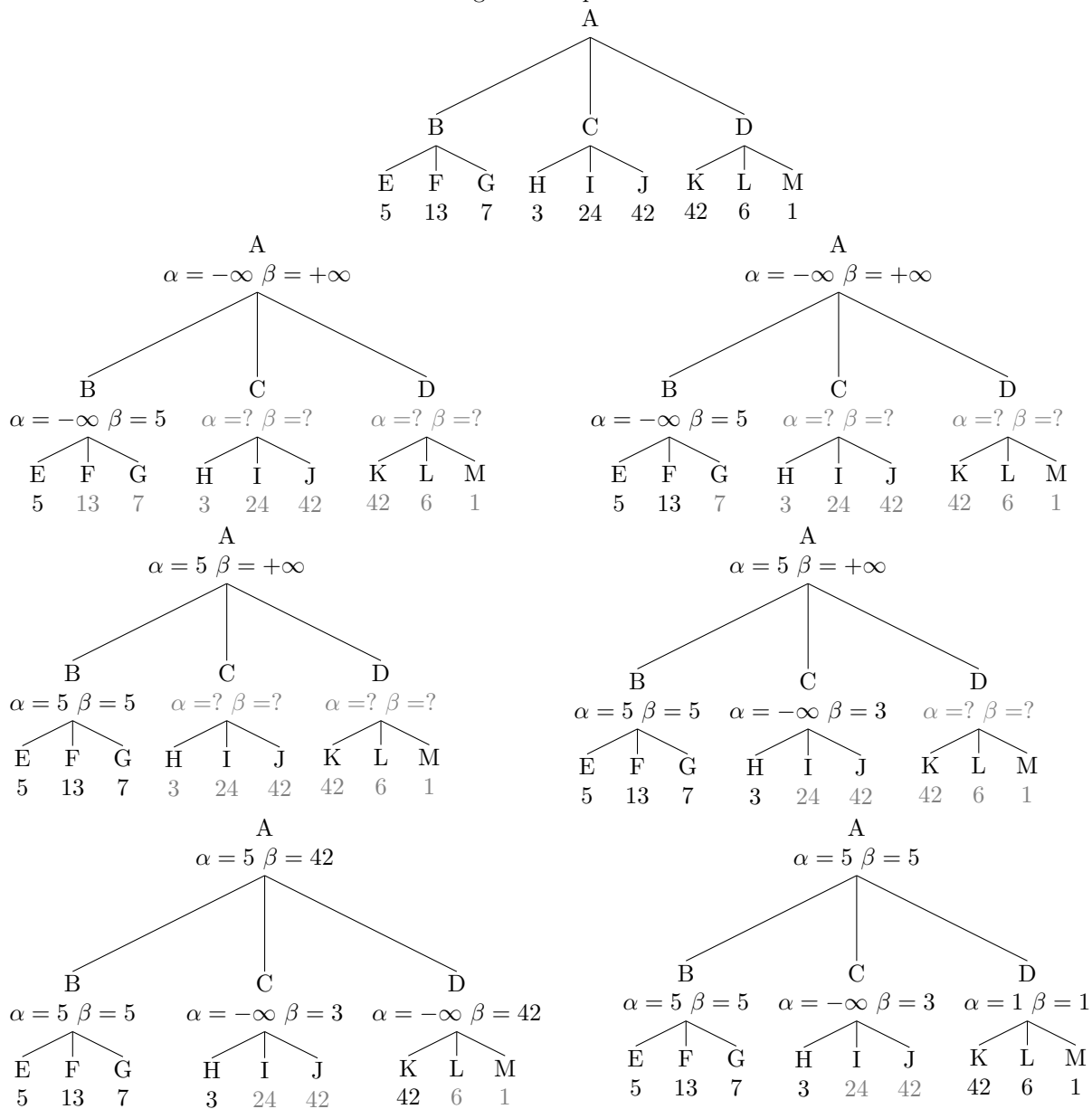
Upper Confidence Bound

[?] haben gezeigt, dass es eine Strategie, von ihnen Upper Confidence Bound 1 (UCB1) genannt, gibt, die ein logarithmisches Wachstum des Regretts über n ermöglicht, ohne dass dazu weitere Informationen bezüglich der Gewinnverteilung bekannt sein müssen, sobald die Belohnungen zwischen 0 und 1 liegen. Die Strategie spielt dabei jenen Arm j , der UCB1 maximiert, mit:

$$UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

Dabei ist \bar{X}_j der durchschnittliche Gewinn beim Spielen des Armes j , n_j die Anzahl der Male zu denen j gewählt wurde und n die Anzahl der insgesamt gespielten Durchgänge. Der linke Term ist der durchschnittliche Gewinn und stellt damit die Exploitation sicher, während sich der rechte Term für selten gewählte Arme stetig erhöht und die Exploration sicherstellt.

Abbildung 2.1: Beispielhafter Game Tree



Kapitel 3

Othello

Von Othello gibt verschiedene Varianten. Eine Variante ist Reversi.

Othello wird auf einem 8x8 Spielbrett mit zwei Spielern gespielt. Es gibt je 64 Spielsteine, welche auf einer Seite schwarz, auf der anderen weiß sind. Der Startzustand besteht aus einem leeren Spielbrett, in welchem sich in der Mitte ein 2x2 Quadrat aus abwechselnd weißen und schwarzen Steinen befindet. Anschließend beginnt der Spieler mit den schwarzen Steinen. Die Spielfelder werden in verschiedene Kategorien eingeteilt (siehe Abbildung):

- Randfelder: äußere Felder
- C-Felder: Felder, welche ein Feld horizontal oder vertikal von den Ecken entfernt sind
- X-Felder: Felder, welche ein Feld diagonal von den Ecken entfernt sind
- Zentrum: innerste Felder von C3 bis F6
- Zentralfelder: Felder D4 bis E5

Diese Kategorien sind für die spätere Strategie wichtig.

3.1 Spielregeln

Othello besitzt einfache Spielregeln, welche im Spielverlauf aber auch taktisches oder strategisches Geschick erfordern. Jeder Spieler legt abwechselnd einen Stein auf das Spielbrett. Dabei sind folgende Spielregeln zu beachten:

- Ein Stein darf nur in ein leeres Feld gelegt werden.
- Auf mindestens einer Seite des Steines (vertikal, horizontal oder diagonal) muss ein gegnerischer Stein durch diesen Stein umschlossen werden. Dies bedeutet, dass nach dem angrenzenden gegnerischen Stein beliebig viele generische Steine folgen und durch einen eigenen Stein abgeschlossen werden. Es dürfen sich keine leere Felder dazwischen befinden.
- Die umschlossenen generischen Steine werden umgedreht, sodass alle Steine die eigene Farbe besitzen.
- Ist für beide Spieler kein Zug mehr möglich, ist das Spiel beendet. Der Spieler mit den meisten Steinen seiner Farbe gewinnt das Spiel.

weiter

Abbildung

3.2 Spielverlauf

Das Spiel wird in drei Abschnitte eingeteilt:

- Eröffnungsphase
- Mittelspiel
- Endspiel

Diese Abschnitte sind jeweils 20 Spielzüge lang. Im Eröffnungs- und Endspiel stehen zum Mittelspiel wenige Zugmöglichkeiten zur Verfügung, da entweder nur wenige Steine auf dem Spielbrett existieren oder das Spielbrett fast gefüllt ist und nur noch einzelne Lücken übrig sind. Im Mittelspiel existieren sehr viele Möglichkeiten, da sich schon mindestens 20 Steine auf dem Spielbrett befinden und diese sehr gute Anlegemöglichkeiten bieten.

3.3 Spielstrategien

Wie in anderen Spielen gibt es auch in Othello verschiedene Strategien. Dabei kann beispielsweise offensiv gespielt werden, indem versucht wird möglichst viele Steine in einem Zug zu drehen. Es gibt auch defensive „stille“ Züge. Ein „stiller“ Zug dreht keinen Frontstein um und dreht möglichst nur wenige innere Steine um (vgl. [?]).

Generell ist eine häufig genutzte Strategie die eigene Mobilität zu erhöhen und die Mobilität des Gegners zu verringern. Mit dem Begriff Mobilität sind die möglichen Zugmöglichkeiten gemeint. Durch das Einschränken der gegnerischen Mobilität hat dieser weniger Zugmöglichkeiten und muss so ggf. strategisch schlechtere Züge durchführen.

Die Position der Steine auf dem Spielbrett sollte ebenfalls nicht vernachlässigt werden. beispielsweise sollen Züge auf X-Felder vermieden werden, da der Gegner dadurch Zugang zu den Ecken bekommt. Dadurch können ggf. die beiden Ränder und die Diagonale gedreht werden und in den Besitz des Gegners gelangen.

In der Eröffnungsphase sollten die Randfelder ebenfalls vermieden werden, da diese in dieser frühen Phase des Spiels noch gedreht werden können und der taktische Vorteil in einen strategischen Nachteil umgewandelt wird.

3.4 Eröffnungszüge

In der nachfolgenden Tabelle ?? sind verschiedene Spieleröffnungen und deren Häufigkeit in Spielen aufgelistet. Spielzüge werden in Othello durch eine Angabe der Position, auf welche der Stein gesetzt wird, dargestellt. Ein vollständiges Spiel lässt sich deshalb in einer Reihe von maximal 60 Positionen darstellen.

Name	Häufigkeit	Spielzüge
Tiger	47%	f5 d6 c3 d3 c4
Rose	13%	f5 d6 c5 f4 e3 c6 d3 f6 e6 d7
Buffalo	8%	f5 f6 e6 f4 c3
Heath	6%	f5 f6 e6 f4 g5
Inoue	5%	f5 d6 c5 f4 e3 c6 e6
Shaman	3%	f5 d6 c5 f4 e3 c6 f3

Tabelle 3.1: Liste von Othelloeröffnungen [?]

[?] gibt folgende weitere Tipps für Eröffnungen:

- Versuche weniger Steinchen zu haben als dein Gegner.
- Versuche das Zentrum zu besetzen.
- Vermeide zu viele Frontsteine umzudrehen.
- Versuche deine Steinchen in einem Haufen zu sammeln statt einige verstreute isolierte Steinchen zu haben.
- Vermeide vor dem Mittelspiel auf die Kantenfelder zu setzen.

Frontsteine sind die äußersten Steine auf dem Spielbrett um das Zentrum. Viele dieser Tipps können auch im späteren Spielverlauf verwendet werden.

Zitatweise ok?

Kapitel 4

Implementierung der KI

Kapitel 5

Evaluierung

Kapitel 6

Fazit

Notes

Literaturverzeichnis

- [ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.
- [BPW⁺12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.
- [CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.
- [OB] George Ortiz and Matthias Berg. Eröffnungsstrategie.
- [RN16] Stuart J. Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Always learning. Pearson, Boston and Columbus and Indianapolis and New York and San Francisco and Upper Saddle River and Amsterdam, Cape Town and Dubai and London and Madrid and Milan and Munich and Paris and Montreal and Toronto and Delhi and Mexico City and Sao Paulo and Sydney and Hong Kong and Seoul and Singapore and Taipei and Tokyo, third edition, global edition edition, 2016.
- [Tok10] Michel Tokic. Adaptive ϵ -greedy exploration in reinforcement learning based on value differences. In Dillmann, Rüdiger and Beyerer, Jürgen and Hanebeck, Uwe D. and Schultz, Tanja, editor, *KI 2010: Advances in Artificial Intelligence*, pages 203–210, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.