Othello

— Entwicklung einer KI für das Spiel —

Patrick Müller, Max Zepnik

5. April 2019

Inhaltsverzeichnis

1 Einleitung

Computergegner ..
. test..

am Ende schreiben

text1 ...

auf Fazit beziehen?

2 Othello

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 2.1):

- Randfelder: äußere Felder (blaue Felder) o.V. 2015
- C-Felder: Felder, welche ein Feld horizontal oder vertikal von den Ecken entfernt sind (vgl. Berg, Matthias o.D.)
- X-Felder: Felder, welche ein Feld diagonal von den Ecken entfernt sind o.V. 2015
- Zentrum: innerste Felder von C3 bis F6 (grüne Felder) o.V. 2015
- Zentralfelder: Felder D4 bis E5 o.V. 2015
- Frontsteine: die äußersten Steine auf dem Spielbrett, welche andere Steine umschließen (vgl. Ortiz, George and Berg, Matthias o.D.).

Diese Kategorien sind für die spätere Strategie wichtig.

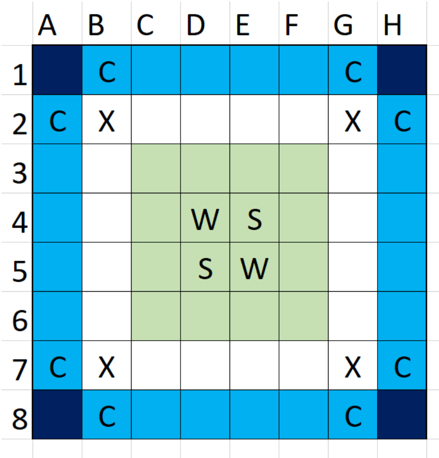


Abbildung 2.1: Kategorien des Spielfeldes

Von Othello gibt verschiedene Varianten. Eine Variante ist Reversi. Die verschiedenen Varianten sind allerdings bis auf die Startposition gleich. Bei der Variante Reversi sind die Zentralfelder noch nicht besetzt und die Spieler setzen die vier Steine selbst, während bei Othello die Startaufstellung fest vorgegeben ist.

2.1 Spielregeln

Othello besitzt wenige 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 welche auch in Abbildung 2.2 abgebildet sind:

- Ein Stein darf nur in ein leeres Feld gelegt werden.
- Es dürfen nur Steine auf Felder gelegt werden, welche einen oder mehrere gegnerischen Steine mit einem bestehenden Stein umschließen würden. Dies ist im linken Spielbrett durch grüne Felder und im mittleren Spielbrett durch das gelbe Feld hervorgehoben. Das gelbe Feld (F5) umschließt mit dem Feld D5 (blau) einen gegnerischen Stein. Es können auch mehrere Steine umschlossen werden. Allerdings dürfen sich dazwischen keine leeren Felder befinden.
- Von dem neu gesetzten Stein in alle Richtungen ausgehend werden die umschlossenen generischen Steine umgedreht, sodass alle Steine die eigene Farbe besitzen. In dem Beispiel ist das im dem rechten Spielbrett zu sehen. F5 umschließt dabei das Feld E5 (rot). Dieses Feld wird nun gedreht und wird schwarz.
- Ist für einen Spieler kein Zug möglich muss dieser aussetzen. Ein Spieler darf allerdings nicht freiwillig aussetzen wenn noch mindestens eine Zugmöglichkeit besteht.
- Der Spieler mit den meisten Steinen seiner Farbe gewinnt das Spiel.
- Ist für beide Spieler kein Zug mehr möglich, ist das Spiel beendet.
- Das Spiel endet ebenfalls, wenn alle Felder des Spielbrettes besetzt sind.

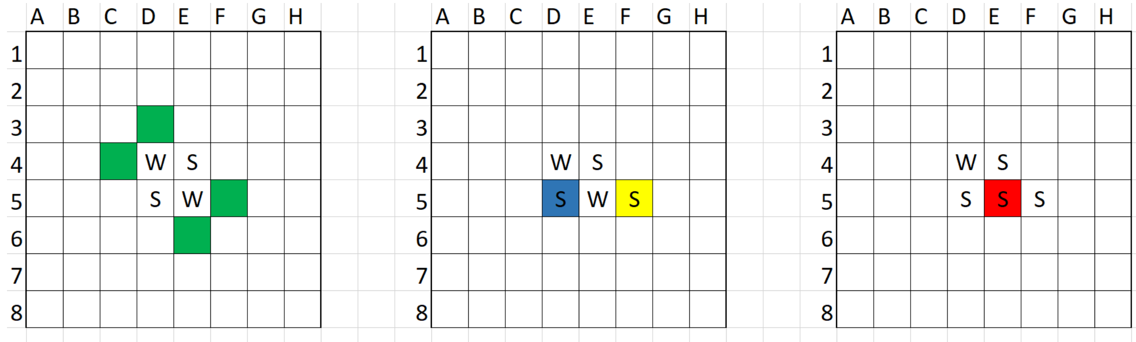


Abbildung 2.2: valide Zugmöglichkeiten für Schwarz und ausgeführter Zug

2.2 Spielverlauf

Das Spiel wird in drei Abschnitte eingeteilt Ortiz, George and Berg, Matthias o.D.:

- Eröffnungsphase
- Mittelspiel
- Endspiel

Diese Abschnitte sind jeweils 20 Spielzüge lang. Im Eröffnungs- und Endspiel stehen im Vergleich 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 viele Möglichkeiten, da sich schon mindestens 20 Steine auf dem Spielbrett befinden und diese sehr gute Anlegemöglichkeiten bieten.

2.3 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. Ortiz, George and Berg, Matthias o.D.).
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.

2.4 Eröffnungszüge

In der nachfolgenden Tabelle 2.1 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 2.1: Liste von Othelloeröffnungen Ortiz, George and Berg, Matthias o.D.

Ortiz, George and Berg, Matthias o.D. 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 eigene Steine in einem Haufen zu sammeln statt diese zu verstreuen.
- Vermeide vor dem Mittelspiel auf die Kantenfelder zu setzen.

Viele dieser Tipps können auch im späteren Spielverlauf verwendet werden.

3 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für eine Künstliche Intelligenz für das Spiel Othello erläutert.
Zunächst werden dazu einige Begriffe der Spieltheorie eingeführt. Diese werden dann zur Beschreibung einiger deterministischer Algorithmen zum treffen einer Spielentscheidung verwendet. Den Abschluss des Kapitels bildet schließlich die Beschreibung einer stochastische Methode für diesen Zweck.

3.1 Spieltheorie

In dem folgenden Unterkapitel werden grundlegende Definitionen eingeführt. Diese sind an Russell und Norvig 2016 angelehnt.

Definition 1 (Spiel (Game)(vgl. Russell und Norvig 2016 S. 162)) Ein Spiel besteht aus einem Tupel der Form

$$\mathtt{G} = \langle Q, S_0, \mathtt{player}, \mathtt{actions}, \mathtt{result}, \mathtt{terminalTest}, \mathtt{utility} \rangle$$

Sei Q die Menge aller Zustände im Spiel. Ein Spielzustand besteht aus dem aktuellen Spielbrett, der Zugnummer, dem aktiven Spieler und weiteren Parametern, welche zur genauen Darstellung eines Spielzustands führen.

- $\mathtt{Q}$: Menge aller Spielzustände
- $\mathtt{S}_0 \in \mathtt{Q}$ beschreibt den Startzustand des Spiels.
- $\mathtt{Players}$: Menge aller Spieler: $\mathtt{Players} = \{S, W\}$
- $\mathtt{player} : \mathtt{Q} \to \mathtt{Players}$ ist auf der Menge der Spielzustände definiert und gibt den aktuellen Spieler p zurück.
- $\mathtt{actions} : \mathtt{Q} \to 2^{\mathtt{moves}}$ gibt die Menge der validen Zugmöglichkeiten eines gegeben Zustands zurück.
  $\mathtt{moves}$ ist die Menge aller Zugmöglichkeiten.

$\texttt{moves} = \{< X, Y > |X \in \{0..7\} \wedge Y \in \{0..7\}\}$
Diese geben die Koordinate der Zugposition des neuen Steines auf dem Spielbrett an.

- $\texttt{result} : \texttt{Q} \times \texttt{actions} \rightarrow \texttt{Q}$ definiert das Resultat einer durchgeführten Aktion a und in einem Zustand s.
- $\texttt{terminalTest} : \texttt{Q} \rightarrow \mathbb{B}$ prüft ob ein Zustand s ein Terminalzustand ($s \in \texttt{terminalStates}$), also Endzustand, darstellt.
  $$\texttt{terminalTest}(s) = \begin{cases} \texttt{True} \ |s \in \texttt{terminalStates} \\ \texttt{False} \ |s \notin \texttt{terminalStates} \end{cases}$$
- $\texttt{utility} : \texttt{terminalStates} \times \texttt{Players} \rightarrow \{-1, 0, 1\}$ gibt einen Zahlenwert aus den Eingabenwerten
  s (Terminalzustand) und p (Spieler) zurück.
  Positive Werte stellen einen Gewinn, negative Werte einen Verlust dar. „0“ stellt ein Unentschieden dar.

Eine spezielle Art von Spielen sind Nullsummenspiele.

Definition 2 (Nullsummenspiele (vgl. Russell und Norvig 2016 S. 161)) In einem Nullsummenspiel ist die Summe der utility Funktion eines Endzustandes (terminalState) über alle Spieler 0. Es gilt also:

$$\forall s \in \texttt{terminalStates} : \sum_{p \in Players} utility(s, p) = 0$$

In Othello spielen zwei Spieler gegeneinander. Es gibt also nur die drei Möglichkeiten:

- Weiß gewinnt, Schwarz verliert
- Schwarz gewinnt, Weiß verliert
- Unentschieden

Durch den Startzustand $\texttt{S}_0$ und der Funktion actions und result wird ein Spielbaum (Game Tree) aufgespannt.

Definition 3 (Spielbaum (Game Tree)(vgl. Russell und Norvig 2016 S. 162)) Ein Spielbaum besteht aus einer
Wurzel, welche einen bestimmten Zustand (Startzustand $\texttt{S}_0$) 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 Zustand `s` zum Kindknoten `s'` zu gelangen. Diese Kindknoten können wiederum weitere Knoten enthalten oder ein Endpunkt des Baumes (Blatt = Spielende) darstellen. Die mathematische Definition wird rekursiv durchgeführt:

umformulieren ?

$\texttt{Spielbaum} : \texttt{Q} \times \texttt{List}(\texttt{actions}) \times \texttt{List}(\texttt{Spielbaum})$
$\texttt{Node}(S, M, T) \in \texttt{Spielbaum}$ g.d.w:

- $S \in Q \cup \{\Omega\}$
- $M = [m_1, .., m_n] \in \texttt{List}(\texttt{actions})$, wobei gilt: $\forall i \in \{1..n\} : m_i \in \texttt{actions}(S)$
- $T = [t_1, .., t_n] \in \texttt{List}(\texttt{Spielbaum})$, wobei gilt: $\forall i \in \{1..n\} : t_i = \texttt{result}(S, m_i)$

Definition 4 (Suchbaum (Search Tree)(vgl. Russell und Norvig 2016 S. 163)) Ein Suchbaum ist ein Teil des Spielbaums. Die Wurzel des Spielbaums besteht aus dem aktuellen Spielzustand. Alle Kindknoten und Blätter entsprechen dem Spielbaum beginnend ab dem aktuellen Zustand. Es gilt also:
$B \in A \wedge B = A[s]$ mit $A \in \texttt{Spielbaum}, B \in \texttt{Suchbaum}(s), s \in Q$

Überleitung einfügen

3.2 Spielstrategien

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

3.2.1 MiniMax

Die erste hier erläuterte Strategie ist der MiniMax Algorithmus. Dieser ist folgendermaßen definiert (siehe Russell und Norvig 2016 S.164):

$$\texttt{MiniMax}(s,p) = \begin{cases} \texttt{Utility}(s,p);\ \text{wenn TerminalTest(s)} \\ \texttt{max}(\{\texttt{MiniMax}(\texttt{result}(s,a) \mid a \in \texttt{actions}(s)\}, (p+1)\%2);\ \text{wenn Spieler am Zug} \\ \texttt{min}(\{\texttt{MiniMax}(\texttt{result}(s,a) \mid a \in \texttt{actions}(s)\}, (p+1)\%2);\ \text{wenn Gegner am Zug} \end{cases}$$

$$\texttt{getBestMove} : Q \times \{-1,0,1\} \to \texttt{actions}$$

$$\texttt{getBestMove}(s, \texttt{score}, p) = \{a \mid a \in \texttt{actions} \wedge \texttt{MiniMax}(\texttt{result}(s,a), (p+1)\%2) == \texttt{score}\}.\texttt{any}()$$

Die Funktion `getBestMove` ermittelt eine Menge aller Züge, welche den von `MiniMax` zurückgegeben Wert besitzen und wählt aus dieser Menge einen Zug aus.
Der Spieler sucht durch diese Funktion den bestmöglichen Zug aus den verfügbaren Zügen (`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 Suchbaumes voraus. Bei einem durchschnittlichen Verzweigungsfaktor von f bei einer Tiefe von d resultiert daraus eine Komplexität von $\mathcal{O}(d^{\mathrm{f}})$. 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 MiniMax-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.

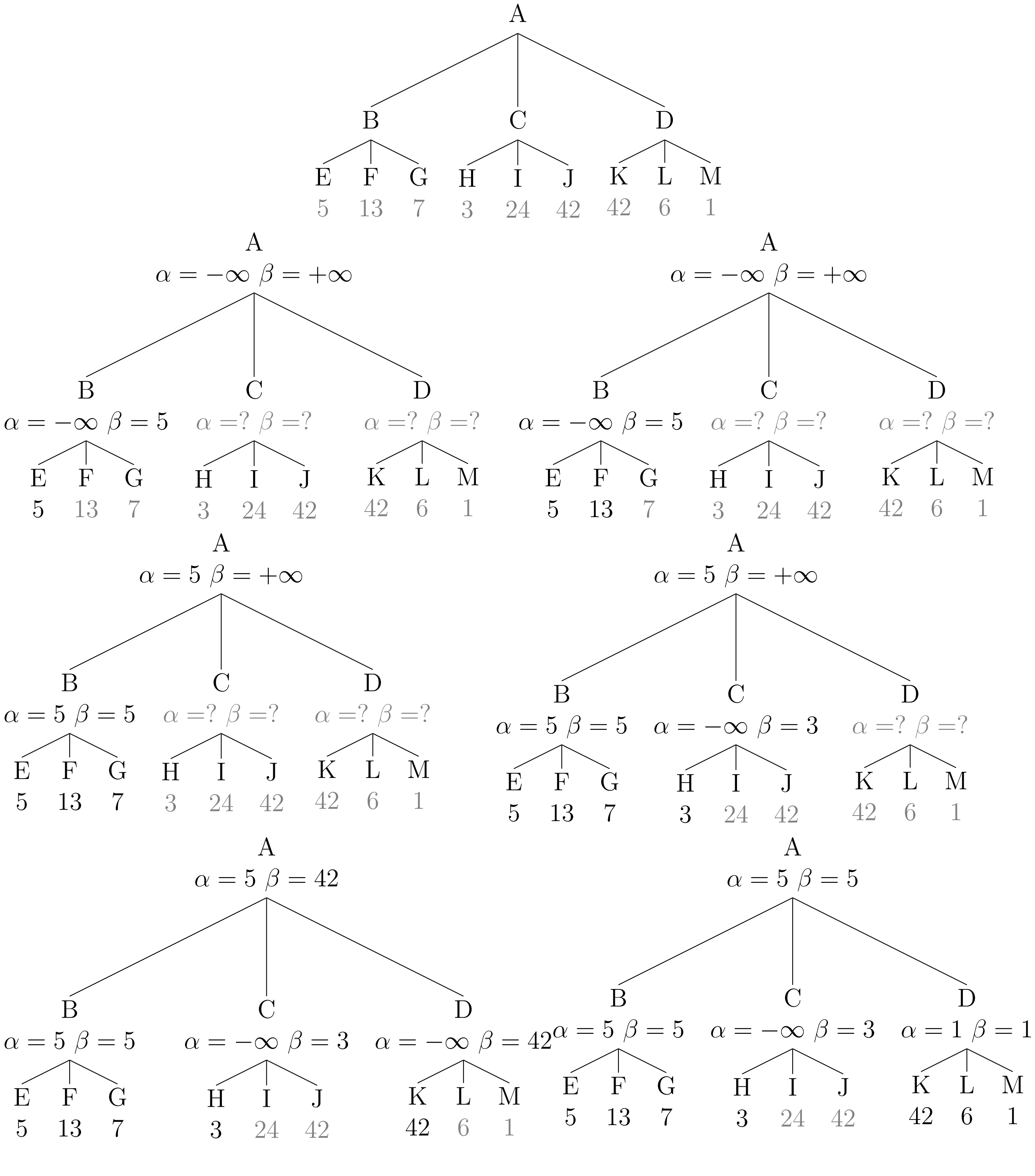
3.2.2 Alpha-Beta Abscheiden

Der MiniMax Algorithmus berechnet nach dem Prinzip „depth-first“ stets den kompletten Spielbaum.
Bei der Betrachtung des Entscheidungsverhaltens des Algorithmus fällt jedoch schnell auf, dass ein nicht unerheblicher Teil aller möglichen Züge durch einen menschlichen Spieler gar nicht erst in Betracht gezogen wird. Dies geschieht aufgrund der Tatsache, dass diese Züge in einem schlechteren Ergebnis resultieren würden als ein anderer, letztendlich ausgewählter, Zug.
Dem Alpha-Beta Abschneiden (Alpha-Beta Pruning) Algorithmus liegt der Gedanke zugrunde, dass die
Zustände, die in einem realen Spiel nie ausgewählt 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 Russell und Norvig 2016 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 Spielbaum 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 Warum
werden.
Im nachfolgenden Spielbaum 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.
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

Abbildung 3.1: Beispielhafter Spielbaum



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 einer Invariante des Alpha-Beta Abschneiden Algorithmus angegeben (siehe Listing 3.1). Es handelt sich um eine angepasste Version von Stroetmann 2019

Quelltext 3.1: Pseudoimplementierung von Alpha-Beta Abschneiden

```
alphaBeta(State, player, alpha = -1, beta = 1) {
        if (finished(State)) {
            return utility(State, player)
        }
        val := alpha
        for (ns in nextStates(State, player)) {
            val = max({ val, -alphaBeta(ns, other(player), -beta, -alpha) })
            if (val >= beta) {
                return val
            }
            alpha = max({ val, alpha })
```

```
        }
        return val
}
```

Bei der angegebenen Implementierung handelt es sich um eine rekursive Umsetzung. Nachfolgend sei das Programm erleutert:

1. Im Basisfall wurde bereits ein Blatt des Spielbaumes erreicht. Damit ist das Spiel bereits beendet. In diesem Fall kann mit *utility* der Wert des Zustands *State* für den entsprechenden Spieler *player* zurückgegeben werden.

2. In der Variable *val* wird der maximale Wert aller von *State* erreichbaren Zustände, sofern *player* einen Zug ausführt, gespeichert.
Da der Algorithmus per Definition alle Wertigkeiten kleiner *alpha* ausschließen soll kann die Variable mit *alpha* initialisiert werden.

3. Nun wird über alle Folgezustände *ns* aus der Menge $nextStates(State, player)$ iteriert.

4. Nun wird rekursiv jeder Zustand *ns* ausgewertet. An dieser Stelle ist jedoch der andere Spieler an der Reihe. Entsprechend erfolgt dies für den anderen Spieler. Da es sich um ein Nullsummenspiel handelt ist der Wert eines Zustandes aus Sicht des Gegners von *player* genau der negative Wert der Wertigkeit für *player*. Aus diesem Grund müssen die Rollen von *alpha* und *beta* vertauscht und außerdem die Vorzeichen invertiert werden.

5. Da laut der Spezifikation des Algorithmus nur die Wertigkeit von Zuständen berechnet werden sollen in denen diese kleiner oder gleich *beta* ist, wird die Auswertung aller Folgezustände mit einem *val* der größer oder gleich *beta* ist abgebrochen. In diesem Fall wird *val* zurückgegeben.

6. Sobald ein Folgezustand mit einem größeren Wert als *alpha* gefunden wurde kann *alpha* auf den entsprechenden Wert erhöht werden. Sobald klar ist, dass der Wert *val* erreicht werden kann, so sind Werte kleiner als *val* nicht mehr relevant.

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. In Kapitel 3.2.3 werden unterschiedliche Heuristiken erklärt.

3.2.3 Suboptimale Echtzeitentscheidungen

Selbst die gezeigten Verbesserung des MiniMax-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 Sekunden, getroffen werden soll. Hinzu kommt die Tatsache, dass viele Spiele unter Verwendung von derzeit erhältlicher Hardware (noch) nicht lösbar sind. 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 versucht den Wert eines Spielzustandes anhand einzelner Eigenschaften des Zustandes anzunähern. Wie in Kapitel 2.3 erläutert, hat ein Spieler der eine Ecke des Feldes besetzt in der Regel einen Vorteil. Ein solcher Zustand würde durch die Heuristik entsprechend besser bewertet werden.
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. Jedoch wird die Berechnung der Heuristik mit steigender Genauigkeit meist komplizierter und somit auch rechen- und damit zeitintensiver. Aus diesem Grund muss immer eine Abwägung aus Genauigkeit und Geschwindigkeit vorgenommen werden.

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.

Vorwärtsabschneiden

Vorwärtsabschneiden (Forward Pruning) durchsucht nicht den kompletten Spielbaum, sondern durchsucht nur einen Teil. Eine Möglichkeit ist eine Strahlensuche, welche nur die „besten“ Züge durchsucht (vgl. Russell und Norvig 2016 S. 175). Die Züge mit einer geringen Erfolgswahrscheinlichkeit werden abgeschnitten und nicht bis zum Blattknoten evaluiert. Durch die Wahl des jeweiligen Zuges mit der höchsten Gewinnwahrscheinlichkeit können aber auch sehr gute bzw. schlechte Züge nicht berücksichtigt werden, wenn diese eine geringe Wahrscheinlichkeit besitzen. Durch das Abschneiden von Teilen des Spielbaum wird die Suchgeschwindigkeit deutlich erhöht. Der in dem Othello-Programm „Logistello“ verwendete „Probcut“ erzielt außerdem eine Gewinnwahrscheinlichkeit von 64% gegenüber der ursprünglichen Version ohne Vorwärtsabschneiden (vgl. Russell und Norvig 2016 S. 175).

Suche gegen Nachschlagen

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 Folgezustände. 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. Das Mittelspiel jedoch hat zu viele Zugmöglichkeiten, um eine Tabelle der möglichen Spielzüge bis zum Spielende aufstellen zu können.Den in diesem Spielabschnitt existieren üblicherweise 4 bis 12 Zugmöglichkeiten. Der in dem Kapitel 2.4 werden die bekanntesten Eröffnungsstrategien aufgelistet.

Viele Spielstrategien wie beispielsweise die MiniMax-Strategie setzen den kompletten oder wenigstens einen großen Teil des Spielbaums voraus. Dieser kann entweder berechnet werden oder aus einer Lookup-Tabelle gelesen werden. Je nach Verzweigungsfaktor der einzelnen Spielzüge kann diese allerdings sehr groß sein. Selbst im späten Spielverlauf gibt es verschiedene Spiele, welche einen großen Spielbaum besitzen.
Beispielsweise existieren für das Endspiel in Schach mit einem König, Läufer und Springer gegen einen König 3.494.568 mögliche Positionen (vgl. Russell und Norvig 2016 S.176).
Dies sind zu viele Möglichkeiten um alle speichern zu können, da noch sehr viel mehr Endspiel-Kombinationen als diese existieren.
Anstatt die Spielzustände also zu speichern, können auch die verbleibenden Spielzustände berechnet werden. Othello besitzt gegenüber Schach den Vorteil, dass die Anzahl der Spielzüge auf 60 bzw. 64 Züge begrenzt sind. Dadurch kann in der Endphase des Spiel ggf. der komplette verbleibende Spielbaum berechnet werden, da die Anzahl der möglichen Zugmöglichkeiten eingeschränkt wird.
Bei der Berechnung der Spielzüge sind die Suchtiefe und der Verzweigungsfaktor entscheidend für die Berechnungsdauer. Aus diesem Grund können im Mittelspiel keine MiniMax-Algorithmen bis zu den Blattknoten des Spielbaumes ausgeführt werden, da die Menge des benötigten Speicherplatzes außerhalb jeglicher Grenzen eines Arbeits- oder Gamingscomputers liegen.

3.3 Monte Carlo Algorithmus

Im Gegensatz zu den bisher gezeigten Algorithmen verwendet der Monte Carlo Algorithmus einen Stochastischen Ansatz um einen Zug auszuwählen. Im nachfolgenden Abschnitt wird die Funktionsweise des Monte Carlo Algorithmus erklärt. Daran angeschlossen folgen Möglichkeiten Strategische Überlegungen zum Spiel Othello einzubringen. Dabei sind die nachfolgenden Ausführungen stark angelehnt an jene von Nijssen 2007.

3.3.1 Algorithmus

Als Ausgangspunkt legt der Monte Carlo Algorithmus die Menge der Züge zugrunde, die ein Spieler unter Wahrung der Spielregeln wählen kann. Diese Züge seien nachfolgend Zug-Kandidaten genannt. Enthält die Menge keine Züge, so bleibt dem Spieler nichts anderes übrig

als auszusetzen. Enthält die Menge nur einen Zug, so muss der Spieler diesen ausführen. Per Definition ist dies dann der bestmögliche Zug. Enthält die Menge hingegen mindestens zwei mögliche Züge, so gilt es den besten unter ihnen auszuwählen. Um den besten Zug zu ermitteln, wird das Spiel mehrfach, die Anzahl sei N_P, bis zum Ende simuliert. Während der Simulation wird jeder mögliche Zug gleich häufig gewählt. Der Rest des simulierten Spiels wird dann zufällig zu Ende gespielt. Sobald alle Durchgänge erfolgt sind, wird das Durchschnittliche Ergebnis für jeden möglichen Zug berechnet. Dabei gibt es zwei Möglichkeiten dieses Ergebnis zu berechnen:
Wahlweise kann die Durchschnittliche Punktzahl eines Zuges oder die durchschnittliche Anzahl an gewonnenen Spielen herangezogen werden. Jener Zug, der nun das beste Ergebnis verspricht wird gespielt.

3.3.2 Überlegungen zu Othello

Bisher spielt der Algorithmus auf gut Glück ohne sich jeglicher Informationen des Spiels zu bedienen. In der Hoffnung das Spiel des Algorithmus zu verbessern, werden nun weitere Informationen zu Othello herangezogen. Hier sein zwei Möglichkeiten beschrieben um dies zu erreichen:

Vorverarbeitung Wie in entsprechenden Kapitel gezeigt, gibt es strategisch gute und strategisch eher schlechte Züge. In seiner Reinform betrachtet der Monte Carlo Algorithmus jedoch beide Arten von Zügen gleich stark. Die Idee der Methode der Vorverarbeitung besteht darin, schlechte Züge in einem Vorverarbeitungsschritt auszuschließen um diese in den Simulationen gar nicht erst zu spielen. Um zu entscheiden, welche Züge ausgeschlossen werden, werden die einzelnen Spielzustände nach Ausführung des Zuges bewertet. Dazu werden die im entsprechenden Kapitel beschriebenen Kategorien von Spielsteinen herangezogen und mit einem entsprechenden Punktwert belegt. Für die Entscheidung an sich kann nun zwischen zwei Strategien gewählt werden:
Entweder kann eine feste Anzahl, diese sei N_S, an best bewertesten Züge ausgewählt werden oder alternativ eine variable Anzahl. Dies geschieht in dem der Durchschnitt aller Bewertungen bestimmt wird und nur jene Züge ausgewählt werden die eine gewisse Bewertung relativ zum Durchschnittswert haben. Dies wird in einer prozentualen Erfüllung des Durchschnittswertes, diese sei p_s, angegeben.

Pseudozufällige Zugauswahl In der Standardversion des Monte Carlo-Algorithmus werden die simulierten Spiele nach der Wahl des ersten Zuges zufällig zu Ende gespielt. Dem hier beschriebenen Ansatz liegt die Idee zu Grunde auch in dieser Phase der Simulation einige Züge anderen gegenüber zu bevorzugen. Dies geschieht nach dem gleichen Prinzip wie im Abschnitt zur Vorverarbeitung beschrieben. Da es jedoch sehr zeitaufwändig ist, die Bewertung jedes einzelnen Spielzustandes innerhalb der Simulationen vorzunehmen, erfolgt dies nur bis zu einer bestimmten Tiefe. Diese sei N_d.

4 Implementierung der KI

In den folgenden Unterkapiteln werden verschiedene Spielalgorithmen vorgestellt und implementiert. Anschließend werden diese verbessert und auch unter Berücksichtigung des Laufzeitverhaltens analysiert.
Zunächst wird aber die grundsätzliche Programmstruktur erläutert und das Spielgerüst implementiert, damit unterschiedliche Spieler es ausführen können.

4.1 Grundlegende Spiel-Elemente

Die Python Implementierung befindet sich im Verzeichnis „python“ des zu diesem Projekt gehörenden Git Repository. Um das Spiel zur Ausführung zu bringen werden die Pakete „numpy“ sowie „pandas“ benötigt. Sind diese Abhängigkeiten vorhanden, kann das Spiel durch das Ausführen des Kommandos „`python main-game`“ gestartet werden.
Die einzelnen Komponenten wurden unter thematischen Gesichtspunkten in verschiedenen Dateien organisiert. Nachfolgend wird auf die einzelnen Dateien und deren Funktion kurz eingegangen.

4.1.1 Der Spielablauf

In „main-game.py“ wird das Rahmenprogramm gestartet. In diesem werden zunächst die Spieler festgelegt. Anschließend wird ein Spiel erstellt, initialisiert und das Spielbrett ausgegeben (siehe Listing 4.1 Z. 2-4). Die Methode „`game_is_over`“ gibt bei Spielende „`True`“, ansonsten „`False`“ zurück. In der Schleife von Zeile 5 bis Zeile 11 wird der Agent des aktuelle Spielers ermittelt (Z. 6f.). Für diesen wird die Funktion „`get_move`“ aufgerufen. Diese gibt den nach der Strategie des jeweiligen Agenten besten Spielzug zurück. Je nach Spielagent werden unterschiedliche Algorithmen zur Ermittlung dieses Zuges verwendet. Dieser Zug wird gespielt und auf das Spielbrett angewendet (Z. 9). Abschließend wird das aktualisierte Spielbrett und der zuletzt durchgeführte Zug ausgegeben (Z. 10f.). Die Schleife wird bis zum Spielende wiederholt. Danach wird der Gewinner und die gesamte Spieldauer ermittelt und ausgegeben (Z. 12-15).

Quelltext 4.1: Spielablauf in „main-game.py“

```
players = {PLAYER_ONE: player_one, PLAYER_TWO: player_two}
game = Othello()
game.init_game()
game.print_board()
while not game.game_is_over():
    current_player = game.get_current_player()
    player_object = players[current_player]
    move = player_object.get_move(game)
    game.play_position(move)
    game.print_board()
    print(f"Played position: ({COLUMN_NAMES[move[1]]}{move[0] + 1})")
duration = time.time() - start
print("Game is over")
print(f"Total duration: {duration} seconds")
print(f"Winner is {PRINT_SYMBOLS[game.get_winner()]}")
```

4.1.2 Die Klasse „Othello“

Die Klasse „Othello“ modelliert einen Spielzustand und enthält die Grundlegende Spiellogik wie bspw. die Berechnung erlaubter Züge. Nachfolgend wird auf die Art der Speicherung eines Spielzustandes und auf die wichtigsten Funktionen dieser Klasse eingegangen.

Klassenvariablen In Listing 4.2 sind alle Klassenvariablen, sowie deren Initialisierungswerte angegeben.

1. „`_board`“: Bei „`_board`“ handelt es sich um eine Liste von Listen, zur Modellierung der zweidimensionalen Struktur des Spielbretts. Initialisiert wird das Spielbrett in seinem Leerzustand. Daher wird zu Beginn jedes Feld auf „`0`“ zur Repräsentation des leeren Feldes gesetzt.
2. „`_current_player`“: Speichert den Spieler, der im modellierten Spielzustand an der Reihe ist.
3. „`_last_turn_passed`“ wird verwendet um zu speichern ob der vorherige Spieler passen musste. Dadurch kann das Spiel sobald zwei Spieler unmittelbar nacheinander passen müssen beendet werden.

4. „`_game_is_over`“ wird auf „`True`“ gesetzt sobald das Spiel beendet ist
5. „`_fringe`“: In „`_fringe`“ werden alle Felder des Spielfeldes gespeichert die in eine Richtung unmittelbar neben einem bereits besetzten Feld liegen. Durch die Mitführung dieser Information muss zur Berechnung der erlaubten Züge nicht jedes mal über das Spielfeld iteriert werden um zunächst die infrage kommenden Felder zu ermitteln.
6. „`_turning_stones`“: Enthält als Schlüssel alle erlaubten Züge und als Wert jeweils eine Liste jener Spielsteine die durch ausführen des Zuges umgedreht werden. Da zur Ermittlung der erlaubten Züge diese Information bereits berechnet werden muss, wird sie in Form des Dictionarys vorgehalten um diese an anderer Stelle nicht erneut berechnen zu müssen.
7. „`_taken_moves`“: Speichert alle ausgeführten Züge die erforderlich waren um den modellierten Spielzustand zu erreichen, da einige Algorithmen diese Information benötigen.
8. „`_turn_nr`“: Speichert die Nummer des aktuellen Spielzuges, da die verwendete Strategie bei einigen Algorithmen davon abhängt, wie weit das Spiel schon fortgeschritten ist.

Quelltext 4.2: Klassenvariablen der Klasse „Othello“

```
_board = [[0 for _ in range(8)] for _ in range(8)]
_current_player = None

_last_turn_passed = False
_game_is_over = False

_fringe = set()
_turning_stones = dict()

_taken_moves = dict()
_turn_nr = 0
```

Die Funktion „`_compute_available_moves`“ ist in Listing 4.3 angegeben und wird verwendet um die Inhalte des Dictionaries „`_turning_stones`“ zu berechnern.
Da beiden Spielern in der Regel nicht die gleichen Züge zur Verfügung stehen muss zunaächst der vorherige Inhalt von „`_turning_stones`“ gelöscht werden. Dies geschieht in Zeile 2, indem die Datenstruktur neu initialisiert wird.

In Zeile 3 wird eine lokale Referenz des zur Darstellung eines durch den aktuellen Spieler besetzen Feldes verwendeten Symbols erzeugt.
Mit der in Zeile 4 beginnenden Schleife wird über alle in Frage kommenden Züge in der Menge „`_fringe`“ iteriert um zu ermitteln, ob dieser Zug erlaubt wäre.
Dazu wird zunächst eine lokale Menge initialisiert um die durch spielen dieser Position gedrehten Steine zu speichern (Zeile 5).
Nun muss ausgehend von der derzeit betrachteten Position ermittelt werden, ob in irgendeine Richtung Spielsteine gedreht werden würden. Dies erfolgt durch die in Zeile 6 beginnende Schleife.
Dazu wird zunächst das nächste Feld in diese Richtung unter Verwendung einer Hilfsfunktion ermittelt (Zeile 7) und anschließend eine weitere termporäre Menge der in dieser Richtung gedrehten Steine initialisiert (Zeile 8)
Die entsprechende Richtung muss nun solange weiter verfolgt werden, wie ein weiterer Nachbar in diese Richtung vorhanden ist. Dies geschieht durch die in Zeile 9 beginnedne Schleife.
Da in den nachfolgenden Schritten ermittelt werden muss, welcher Spieler das derzeit betrachtete Feld besetzt hat, werden die Indizes der derzeitigen ausgepackt (Zeile 10) und dann verwendet um zu ermitteln welchen Wert das Feld detzeit hat (Zeile 11).
Nun gibt es drei mögliche Fälle:

1. Es befindet sich kein Stein auf dem derzeit betrachteten Feld. In diesem Fall wird die Abfrage in Zeile 12 positiv ausgewertet und diese Richtung muss nicht weiter verfolgt werden. Entsprechend wird die while-Schleife in Zeile 13 abgebrochen.

2. Das derzeit betrachtete Feld wird durch den anderen Spieler besetzt. In diesem Fall ist die Abfrage in Zeile 14 positiv. Da der Stein ggf. umgedreht werden würde, wird die aktuelle Position gespeichert (Zeile 15)

3. Das derzeit betrachtete Feld wird durch den Spieler selbst besetzt. In diesem Fall wird die Abfrage in Zeile 16 positiv ausgewertet. Nun werden alle Steine zwischen der Ausgangsposition und der derzeitigen gedreht. Daher wird die Menge der durch diesen Zug gedrehten Steine mit der in diese Richtung befindlichen Steine vereinigt (Zeile 17) und die Schleife verlassen (Zeile 18).

In Zeile 19 wird das nächste Feld in diese Richtung berechnet.
Gemäß der Regeln muss durch jeden Zug mindestens ein Stein gedreht werden. Aus diesem Grund wird nun ermittelt, ob dies bei diesem Zug gegeben wäre (Zeile 20). Ist dies der Fall, so

werden die gedrehten Steine für diesen Zug in „`_stones_to_turn`“ gespeichert (Zeile 21).
Hat ein Spieler nun keine Möglichkeiten einen Zug durchzuführen, so sind in „`_stones_to_turn`“ keine Züge enthalten. Dieser Fall muss besonders behandelt werden. Tritt er ein, so wird die Abfrage in Zeile 22 positiv ausgewertet.
In diesem Fall muss nochmal unterschieden werden, ob der vorherige Spieler ebenfalls keinen Zug zur Auswahl hatte (Zeile 23).
Falls ja so ist das Spiel zu ende. Dies wird durch das Setzen von „`_game_is_over`“ gespeichert (Zeile 24).
Falls nein (Zeile 25), so wird gespeichert, dass der Spieler passen musste (Zeile 26) und der nächste Zug vorbereitet (Zeile 27).
Hat der Spieler hingegen eine Zugmöglichkeit (Zeile 28) so hat er aus Sicht des folgenden Zuges nicht passen müssen. Entsprechend wird „`_last_turn_passed`“ wieder auf „`False`“ gesetzt.

Quelltext 4.3: Die Funktion „`_compute_available_moves`“

```
1  def _compute_available_moves(self):
2      self._turning_stones = dict()
3      own_symbol = self._current_player
4      for current_position in self._fringe:
5          position_turns = set()
6          for direction in DIRECTIONS:
7              next_step = Othello._next_step(current_position, direction)
8              this_direction = set()
9              while next_step is not None:
10                 (current_x, current_y) = next_step
11                 current_value = self._board[current_x][current_y]
12                 if current_value == EMPTY_CELL:
13                     break
14                 elif current_value != own_symbol:
15                     this_direction.add(next_step)
16                 elif current_value == own_symbol:
17                     position_turns = position_turns | this_direction
18                     break
19                 next_step = Othello._next_step(next_step, direction)
20         if len(position_turns) > 0:
21             self._turning_stones[current_position] = position_turns
22     if len(self._turning_stones) == 0:
23         if self._last_turn_passed:
24             self._game_is_over = True
25         else:
```

```
            self._last_turn_passed = True
            self._prepare_next_turn()
    else:
        self._last_turn_passed = False
```

Weitere Funktionen der Klasse „Othello“ Die Klasse „Othello“ enthält weitere Funktionen, auf welche hier jedoch nicht im Detail eingegangen werden soll. Dennoch sei hier jeweils kurz deren Verwendungszweck der wichtigsten Funktionen genannt:

1. „`play_position`“ verändert den Spielzustand dahingehend, dass der übergebene Zug, sofern er erlaubt ist, ausgeführt wird, die entsprechenden Steine des Gegners gedreht und dessen Zug vorbereitet wird. Dabei wird auch die „`_fringe`“ entsprechend aktualisiert.
2. „`set_available_moves`“ verändert „`_stones_to_turn`“ dahingehend, dass nur noch übergebene Positionen enthalten sind. Kann damit zum Filtern der erlaubten Züge verwendet werden.
3. „`get_available_moves`“: Gibt die erlaubten Züge zur Verwendung in den Spielerimplementierungen zurück
4. „`other_player`“: Gibt das Symbol des anderen Spielers zurück
5. „`utility`“: Gibt gemäß der Definition eines Spiels 0, 1 oder -1 zurück.
6. „`get_winner`“: Ermittelt den Gewinner des Spiels und gibt ihn zurück.
7. „`get_statistics`“: Gibt die Anzahl der Felder pro Spieler zurück.
8. „`get_current_player`“: Gibt den derzeitigen Spieler zurück.
9. „`game_is_over`“: Gibt zurück ob das Spiel bereits zu Ende ist.
10. „`init_game`“: Bereitet den Start eines Spiels vor indem die initial besetzen Felder entsprechen gesetzt werden. Der beginnende Spieler festgelegt und die „`_fringe`“ vorbereitet, sowie „`_stones_to_turn`“ für den ersten Zug berechnet.

4.1.3 Die übrigen Komponenten

Neben den zuvor detailliert besprochenen finden in der Implementierung noch die folgenden Komponenten Anwendung:

1. Konstanten in der Datei „`constants.py`“. Durch die Verwendung von Konstanten bspw. zur Symbolisierung von welchen Spieler ein Feld besetzt ist, wird einerseits von konkreten Werten abstrahiert und diese sind, sofern erforderlich einfach austauschbar. Andererseits wird eine gewisse Konsistenz, bspw. für Mapping-Funktionen zur Ausgabe, über das ganze Programm hinweg erreicht.

2. Hilfsfunktionen in der Datei „`util.py`“ werden dazu genutzt Werte vom Benutzer abzufragen.

4.2 Heuristiken

Wie im Abschnitt 3.2.3 besprochen werden für einige Algorithmen Funktionen zur Approximation des Wertes eines Spielzustandes benötigt. Im Rahmen dieser Arbeit wurden die folgenden Heuristiken implementiert:

1. Nijssen 2007 Heuristik

2. Stored Monte-Carlo-Heuristik

3. Cowthello Heuristik

Die Implementierung aller Heuristiken befinden sich in der Datei „`heuristics.py`“

4.2.1 Nijssen 2007 Heuristik

Die „Nijssen 2007 Heuristik“ wurde aus Nijssen 2007 übernommen. Ihr liegt die im Kapitel 2 erläuterte Idee zugrunde, die einzelnen Spielfelder in Kategorien einzuteilen und jeder Kategorie einen speziellen Wert zuzuweisen. Die Bewertung des Spielzustandes s aus Sicht eines Spielers `player` ergibt sich dann nach der folgenden Formel: $\texttt{heuristic}(\texttt{player}, S) = \sum_{f \in \texttt{domain}(\texttt{F}(s))} w_f * b_f$ wobei $F(s)$ eine Funktion ist, die für das Spielfeld des Zustands s eine Relation zurückgibt die für jedes Feld angibt durch welchen Spieler es besetzt ist oder ob es leer ist, w_f für das dem Feld zugeordnete Gewicht und $b_f = \begin{cases} 1, \ \texttt{wenn F}(s)[f] = \texttt{player} \\ -1, \ \texttt{wenn F}(s)[f] = \texttt{other(player)} \\ 0, \ \texttt{sonst} \end{cases}$

Für die Gewichte b_f der im Kapitel 2 eingeführten Kategorien vergibt Nijssen die folgenden Gewichte:

1. „`Eck-Felder`“: $+5$
2. „`X-Felder`“: -2
3. „`C-Felder`“: -1
4. „`Zentral-Felder`“: $+2$
5. „`Andere-Felder`“: $+1$

4.2.2 Stored Monte-Carlo-Heuristik

Die „Stored Monte-Carlo-Heuristik“ verwendet eine Datenbank. In dieser Datenbank werden die Anzahl der gespielten und gewonnenen Spielzüge gespeichert. Diese wird in den Unterabschnitten Datenbank und Befüllen der Datenbank erklärt. Darauf aufbauend wird die Funktionsweise der Heuristik in dem Abschnitt Heuristik erläutert.

Datenbank Die Datenbank speichert zu jeder Zugnummer die Gewinnwahrscheinlichkeiten der Spieler bei der Verwendung der Feldkategorien zu einer bestimmten Zugnummer. Sie ist in der Datei „`database_moves.csv`“ im CSV-Format gespeichert. Es existieren zehn Feldkategorien. Das Spielfeld ist symmetrisch zum Mittelpunkt aufgebaut. Dies bedeutet, dass Züge, welche symmetrisch zu anderen Zügen sind, als ein Zug angesehen werden können. In Abbildung 4.1 sind die symmetrischen Felder des Othello-Spielbrettes farblich hervorgehoben. Die unterschiedlichen Farben stellen die unterschiedlichen Feldkategorien dar. Diese sind von Null bis Acht durchnummeriert. Das Zentrum ist mit „X“ markiert. Da das Zentrum bei Spielstart schon besetzt ist, wird diese Feldkategorie nicht in der Datenbank gespeichert. Die restlichen neun Kategorien sind als Spalten in der Datenbank abgebildet.

	a	b	c	d	e	f	g	h
1	0	1	2	3	3	2	1	0
2	1	4	5	6	6	5	4	1
3	2	5	7	8	8	7	5	2
4	3	6	8	X	X	8	6	3
5	3	6	8	X	X	8	6	3
6	2	5	7	8	8	7	5	2
7	1	4	5	6	6	5	4	1
8	0	1	2	3	3	2	1	0

Abbildung 4.1: Symmetrie des Spielfeldes

Die Datenbank besteht aus 60 Zeilen und neun Spalten. Die neun Spalten stellen o.g. Feldkategorien („0“ bis „8“) dar. Die c-te Spalte in der n-ten Zeile stellt die Gewinnwahrscheinlichkeiten des Spieles dar, wenn im n-ten Spielzug ein Feld der c-ten Feldkategorie gespielt wird.
Die mathematische Formel dazu lautet: $database_{n,c} = P(\texttt{win}|\texttt{move}(n) \in c)$, wobei gilt:

- $\texttt{move}(n)$ = Feld des n-ten Spielzuges
- $\texttt{c} \in \{\texttt{Feldkategorie} - \texttt{'X'}\}$: c ist eine Feldkategorie außer das Zentrum („X“)

Jede Zelle in dieser Tabelle enthält ein Tripel der Form
$(\texttt{won\_games\_player\_1}, \texttt{won\_games\_player\_2}, \texttt{total\_played\_games})$.
Die erste Komponente stellt die Anzahl der gewonnen Spiele des ersten Spielers dar, die zweite Komponente speichert die Anzahl der Spiele, welche der zweite Spieler gewonnen hat und die dritte Komponente gibt die Gesamtanzahl der gespielten Spiele dieser Spielkategorie an. Die genaue Funktionsweise wird in Kapitel Befüllen der Datenbank erklärt.
Zwischen den drei Komponenten gibt es folgenden mathematischen Zusammenhang:
$\texttt{won\_games\_player\_1} + \texttt{won\_games\_player\_2} \leq \texttt{total\_played\_games}$

Durch unentschiedene Spiele kann die Gesamtanzahl der Spiele größer als die Summe der gewonnen Spiele der zwei Spieler sein. In Abbildung 4.2 ist der Initialzustand der Datenbank dargestellt (vgl. auch Abbildung 4.1 bzgl. der Feldkategorien). Die Startwerte der Tupel sind jeweils „(0, 0, 0)“, da initial noch keine Spiele gespeichert sind.

Zugnummer	Feldkategorie								
	0	1	2	3	4	5	6	7	8
0	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
1	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
2	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
3	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
4	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
5	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)

Abbildung 4.2: Ausschnitt des Initialzustandes der Datenbank

Befüllen der Datenbank Nachdem die Datenbank im vorherigen Abschnitt initialisiert erstellt wurde, wird diese nun mit Spieldaten aus zufällig gespielten Spielen befüllt. In Listing 4.4 ist das Spielen eines zufälligen Spieles und den Aufruf der Funktion „`update_fields_stats_for_single_game`“ abgebildet. Zunächst wird ein Othello-Spiel initialisiert (Z. 1f.). Durch die Nutzung des „Random“-Agenten wird ein komplettes Spiel durchgeführt (Z. 3f.). Anschließend wird der Gewinner ermittelt (Z. 5) und die Zugreihenfolge ermittelt (Z. 6). Die Funktion „`update_fields_stats_for_single_game`“ verwendet diese beiden Parameter um die Datenbank zu aktualisieren. Die Funktion wird in Listing 4.5 abgebildet.

Quelltext 4.4: Befüllen der Datenbank 1

```
1 g = Othello()
2 g.init_game()
3 while not g.game_is_over():
4     g.play_position(Random.get_move(g))
5 winner = g.get_winner()
6 moves = g.get_taken_mv()
7 self.update_fields_stats_for_single_game(moves, winner)
```

In der Funktion „`update_fields_stats_for_single_game`“ (Listing 4.5 Z. 9-12) wird die Liste der Züge in einzelne Züge aufgeteilt (Z. 10), welche anschließend jeweils eine Spalte in einer Datenbankzeile aktualisieren. Dazu werden die Züge, beispielsweise „a1“, in Zeile 11 in eine Feldkategorie umgerechnet (im Beispiel „0“) und ebenfalls der Methode „`update_field_stat`“ übergeben.

Diese Methode liest die Werte der durch Zugnummer und Feldkategorie festgelegte Zelle aus (Z.2). Je nachdem, welcher Spieler dieses Spiel gewonnen hat, wird die Zahl der gewonnenen Spiele des ersten, zweiten oder keinem Spieler um eins erhöht (Z. 3-6). Abschließend wird die Zahl der insgesamt durchgeführten Spiele inkrementiert (Z.7 dritte Komponente) und in die Datenbank zurückgeschrieben (Z.7).
Die Datenbank wurde mit 140.000 Spielen trainiert, um statistische Abweichungen zu minimieren. Hierbei wirkt das Gesetz der großen Zahlen, das besagt, dass sich die Wahrscheinlichkeiten eines Ereignisses bei sehr vielen Wiederholungen dem Erwartungswert annähert.
Zur Auswertung der Datenbank wurde die Klasse „`Analyse`" in der Datei „`analyse_database.py`" erstellt. Diese liest die Datenbank aus und liefert eine farbige Ansicht der Gewinnwahrscheinlichkeiten eines Spielers jeweils pro Zug auf dem Terminal. Neben dem dargestellten Spielbrett wird auch das Minimum, das Maximum, der Durchschnittswert und die Standardabweichung des dargestellten Spielbrettes berechnet.

Quelltext 4.5: Befüllen der Datenbank 2

```
    def update_field_stat(self, turn_nr, field_type, winner):
        (won_games_pl1, won_games_pl2, total_games_played) = self._data[turn_nr][field_type]
        if winner == PLAYER_ONE:
            won_games_pl1 += 1
        elif winner == PLAYER_TWO:
            won_games_pl2 += 1
        self._data[turn_nr][field_type] = (won_games_pl1, won_games_pl2, total_games_played + 1)

    def update_fields_stats_for_single_game(self, moves, winner):
        for turn_nr in range(len(moves)):
            position = self.translate_position_to_database(moves[turn_nr])
            self.update_field_stat(turn_nr, position, winner)
```

Heuristik Mithilfe der o.g. Datenbank wird eine Heuristik implementiert. Die Methode „`heuristic`“ ist in Listing 4.6 abgebildet. Diese ermittelt die verfügbaren Zugmöglichkeiten und die aktuelle Zugnummer (Z. 2f.). Für jede Zugmöglichkeit wird die Gewinnwahrscheinlichkeit für den übergebenen Spieler in ein Dictionary gespeichert (Z. 6f.). Aus diesem Dictionary wird die höchste Wahrscheinlichkeit ermittelt und zurückgegeben (Z. 9f.).

Quelltext 4.6: Stored Monte-Carlo-Heuristik Funktion

```
1  def heuristic(current_player, game_state: Othello):
2      moves = game_state.get_available_moves()
3      turn_nr = game_state.get_turn_nr()
4      move_probability = dict()
5
6      for move in moves:
7          move_probability[move] = database.db.get_likelihood(move, turn_nr, current_player)
8
9      selected_move = max(move_probability.items(), key=operator.itemgetter(1))[0]
10     return move_probability[selected_move]
```

4.2.3 Cowthello Heuristik

Die „Cowthello Heuristik“ wurde aus o.V. o.J übernommen. Ihr liegt ebenfalls der in Kapitel 2 erläuterte Idee zugrunde, die einzelnen Spielfelder in Kategorien einzuteilen und jeder Kategorie einen speziellen Wert zuzuweisen. Die Heuristik unterscheidet sich von der „Nijssen 2007 Heuristik“ nur in der genaueren Definition der Feldkategorien und unterschiedliche Gewichtungen dieser Felder. Die Gewichte der einzelnen Felder sind in Abbildung dargestellt. Die Unterschiede zur „Nijssen 2007 Heuristik“ bestehen in der detaillierteren Gewichtung beispielsweise der Zentralfelder („1“, „5“ oder „50“) im Gegensatz zu „2“. Die restliche Implementierung ist identisch zur „Nijssen 2007 Heuristik“.

	a	b	c	d	e	f	g	h
1	100	-25	25	10	10	25	-25	100
2	-25	-50	1	1	1	1	-50	-25
3	25	1	50	5	5	50	1	25
4	10	1	5	1	1	5	1	10
5	10	1	5	1	1	5	1	10
6	25	1	50	5	5	50	1	25
7	-25	-50	1	1	1	1	-50	-25
8	100	-25	25	10	10	25	-25	100

Abbildung 4.3: Gewichtung der einzelnen Spielfelder

4.3 Start Tabellen

In dem Grundlagenkapitel 3.2.3 wurden die Vor- und Nachteile von Suche vs. Nachschlagen beschrieben. In diesem Kapitel wird die Implementierung der Starttabellen mit Eröffnungszügen erklärt. Im Folgenden werden Spieltabellen und Startdatenbank synonym verwendet, da diese Tabellen als Matrix in einer Datenbank gespeichert werden.

Für viele Spiele, beispielsweise für Schach, existieren Starttabellen, welche die „besten“ Eröffnungszüge speichern. Für Othello existieren zwar mehrere Spieltabellen, aber nur wenige Eröffnungsspiele. In der Implementierung werden die 77 Eröffnungszüge von Robert Gatliff

o.V. 2005 verwendet. Diese bestehen jeweils aus einer Zugreihenfolge, z.B. „`c4, c3, d3, c5, b2`".
Diese Züge werden im Spiel mit dem aktuellen Spielzustand verglichen. Wenn eine oder mehrere gespeicherte Züge mit dem aktuellen Spielzustand übereinstimmen, wir ein Zug aus den verfügbaren Zügen ausgewählt und gespielt. Die Funktion „`get_available_moves_of_start_tables`" ist in Listing 4.7 abgebildet.

Quelltext 4.7: Befüllen der Datenbank 2

```
1 def get_available_moves_of_start_tables(self, game: Othello):
2   if len(self._start_tables) == 0:
3     self._init_start_tables()
4   turn_nr = game.get_turn_nr()
5   available_moves = []
6   taken_mv = game.get_taken_mvs_text()
7   for game in self._start_tables:
8     turn = 0
9     for move in game:
10      if turn < turn_nr:
11        if taken_mv[turn] != move:
12          break
13      elif move != "i8" or move != "nan": # invalid field
14        available_moves.append(move)
15        break
16      turn += 1
17  available_moves = list(dict.fromkeys(available_moves))
18  if "nan" in available_moves:
19    available_moves.remove("nan")
20  return available_moves
```

Beim ersten Aufrufen der Starttabellen, wird die Datenbank initialisiert (Z. 2f.). Ab Zeile sieben werden alle Zeilen der Datenbank mit der aktuellen Zugreihenfolge des Spiels verglichen (Z. 9ff.). Wenn ein Zug in der bestehenden Zugreihenfolge abweichend von dem aktuellen Eintrag der Startdatenbank ist (Z. 11f.), wird der weitere Vergleich mit diesem Eintrag abgebrochen und mit dem folgenden Eintrag fortgefahren.

Da das Spielbrett symmetrisch zum Mittelpunkt ist, wurden die Eröffnungszüge gespiegelt. Dadurch wurde die Datenbank auf 308 Züge erweitert. Diese ist in der Datei „`start_moves.csv`“ als CSV gespeichert.

Die Agenten „Monte Carlo“ und „Alpha-Beta Pruning“ verwenden in der Standardeinstellung Starttabellen. Wenn beide Agenten gegeneinander spielen, werden die ersten Spielzüge beider Spieler sehr stark beschleunigt. Die Agenten müssen keine Züge berechnen, sondern können, bei verfügbaren Eröffnungszügen, durch das Nachschlagen in der Datenbank und die Auswahl eines Spielzuges sehr viel Spielzeit einsparen. Erst wenn die Datenbank keine passende Zugmöglichkeit mehr enthält, starten die Agenten die Berechnung des besten Zuges.

4.4 Agenten

Beim Start eines Partie stehen dem Nutzer mehrere Agenten zur Auswahl die die Rolle eines Spielers übernehmen können. Die Implementierung zu diesen Agenten befindet sich in Unterverzeichnis „`Agents`“.
Die folgende Agenten stehen dabei zur Auswahl:

1. Human
2. Random Player
3. Monte Carlo
4. Alpha-Beta Pruning

4.4.1 Human Agent

Der „Human Agent“ bzw. menschliche Agent stellt eine Schnittstelle die es einem menschlichen Spieler
ermöglicht eine Spielentscheidung zu treffen. Die Implementierung findet sich in der Datei „`human.py`“
Neben dem Spielfeld bekommt der Nutzer dabei eine Liste aller für Ihn möglichen Züge dargestellt. Durch die Eingabe eines Zuges wird dieser im Spielmodell ausgeführt und der nächste Agent wird aufgerufen. Da das Ziel dieser Arbeit darin besteht eine künstliche Intelligenz zur Wahl der Züge zu entwickeln, soll an dieser Stelle nicht weiter auf diesen Agenten eingegangen sein.

4.4.2 Random

Die Implementierung des Agenten „Random“ befindet sich in der Datei „`random.py`“. Dieser Agent ist die einfachste Form der im Rahmen dieser Arbeit eingesetzten Strategien einer künstlichen Intelligenz. Ihr liegt die Idee zugrunde, dass der Agent einen zufälligen Zug aus der Menge der erlaubten Züge auswählt.
In der mit der Arbeit entwickelten Implementierung ist dies derartig umgesetzt, dass der Agent

die Liste der möglichen Züge aus dem Spielzustand abruft und einen zufälligen Index der Liste auswählt um dann den dort angegebenen Zug zu spielen.

4.4.3 Monte Carlo

Hinter diesem Agenten steht das im Kapitel 3.3 erläuterte Prinzip, der zufällig gespielten Spiele zur Ermittlung des Spielzuges mit der höchsten Gewinnwahrscheinlichkeit. Dabei werden ausgehend von einem derzeitigen Spielzustand N Spiele simuliert in dem für beide Spieler der Agent „Random“ verwendet wird. Dabei wird für den jeweils ersten simulierten Zug gespeichert wie oft dieser durchgeführt wurde und wie häufig dies in einer gewonnenen Simulation resultierte. In dem nicht simulierten Spiel wird dann jener Spielzug ausgeführt, bei dem der Quotient aus der Anzahl der nach spielen dieses Zuges gewonnen Spiele und der Anzahl der simulierten Spiele die mit diesem Zug begonnen wurden, am größten ist.
Nachfolgend wird dieses Prinzip anhand des Quellcodes nochmals erläutert.
Die Spielerklasse „MonteCarlo“ ist in der Datei „python/Agents/monteCarlo.py“ zu finden.

Parameter

Das genaue Verhalten des Agenten kann durch die folgenden Parameter beeinflusst werden:

- „`big_n`“: Die Anzahl N der zufällig gespielten Spiele je Zug
- „`use_start_libs`“: Bool'scher Wert ob Startbibliotheken verwendet werden sollen
- „`preprocessor`“: Der verwendete Vorverarbeiter bzw. Präprozessor. Dabei stehen die folgenden bereits im Kapitel 3.3 besprochenen Varianten zur Auswahl:
  - Der feste Selektivität Präprozessor
  - Der variable Selektivität Präprozessor
  - Kein Präprozessor
- „`preprocessor_parameter`“: Parameter des verwendeten Preprozessors. Je nach Auswahl des Prä-
  prozessors steht dieser Parameter für:

  - Die Anzahl der Züge die durch den Präprozessor gelangen
  - Die Prozentuale Abweichung vom Mittelwert der Wertigkeiten der Züge

- „`heuristic`“: Eine der in Kapitel 4.2 beschriebenen. Wird im Präprozessor zur Bewertung von Spielzuständen herangezogen.
- „`use_multiprocessing`“: Boolscher Wert, der angibt ob die N simulierten Spiele unter Verwendung mehrerer Prozesse durchgeführt werden sollen. Bei Systemen mit mehr als einem Prozessor können damit durch Parallelisierung Geschwindigkeitsvorteile erreicht werden.
- „`use_weighted_random`“: Boolscher Wert, der angibt ob die Züge mit einer höheren Gewinnwahrscheinlichkeit stärker in der Random Funktion gewichtet werden sollen.

Die Präprozessoren

Nachfolgend wird kurz auf die Implementierung der in Kapitel 3.3 beschriebenen Präprozessoren eingegangen.

Der feste Selektivität Präprozessor Die in Listing 4.8 abgedruckte Funktion gibt die Implementierung des Präprozessors mit fester Selektivität an.
Die Funktion erhält einen Spielzustand „`game_state`“, den Parameter „`n_s`“ der Anzahl N_s der Züge, die den Präprozessor passieren und eine Heuristik „`heuristic`“, die zur Bewertung der Spielzüge herangezogen wird. In der sich über die Zeilen 3 bis 5 erstreckenden Anweisungen wird eine nach der Bewertung des einzelnen Zuges gemäß der Heuristik sortierte Liste von Zügen erstellt. In Zeile 6 werden dann nur die ersten „`n_s`“ Züge im Spielzustand „`game_state`“ gesetzt.

Quelltext 4.8: Die Funktion „`preprocess_fixed_slectivity`“

```
1 @staticmethod
2 def preprocess_fixed_selectivity(game_state: Othello, n_s, heuristic):
3  heuristic_values = sorted(
4      MonteCarlo.preprocess_get_heuristic_value(game_state, heuristic=heuristic).items(),
5          key=operator.itemgetter(1))
6  game_state.set_available_moves(heuristic_values[:n_s][0])
```

Der variable Selektivität Präprozessor In Listing 4.9 ist die Implementierung des Präprozessors mit variabler Selektivität angegeben.
Wie der Präprozessor mit fester Selektivität, erhält auch diese Funktion einen Spielzustand „`game_state`“ und eine Heuristik „`heuristic`“. Anders als bei dem oben beschriebenen Präprozessor wird hier jedoch ein Wert „`p_s`“ zur Beschreibung der maximalen prozentualen Abweichung p_s vom Mittelwert der Wertigkeit der Züge übergeben.
Für die Berechnung wid zunächst der Wert der Heuristik für alle Spielzüge berechnet (Zeile 3). Daraufhin wird in der Zeile 5 der durchschnittliche Wert berechnet und mit der Anweisung in den Zeilen 6 und 7 nur jene Züge im Spielzustand gesetzt, die einen Wert größer als p_s multipliziert mit dem durchschnittlichen Wert haben.

Quelltext 4.9: Die Funktion „preprocess_variable_slectivity“

```
@staticmethod
def preprocess_variable_selectivity(game_state: Othello, p_s, heuristic):
    heuristic_value_dict = MonteCarlo.preprocess_get_heuristic_value(game_state, heuristic=heuristic)
    heuristic_values = [v for _, v in heuristic_value_dict.items()]
    average_heuristic_value = sum(heuristic_values) / len(heuristic_values)
    game_state.set_available_moves(
        [m for m, v in heuristic_value_dict.items() if v >= p_s * average_heuristic_value])
```

Die Funktion „get_move“

Eine Funktion „get_move“ wird von allen Agenten bereitgestellt und in der „main-game.py“ zur Auswahl eines Zuges gerufen. Beschrieben wird der Zug durch ein Paar, welches die Koordinaten auf dem Spielbrett darstellt.
Nachfolgend wird die in Listing 4.10 abgedruckte Funktion diskutiert:

Als Parameter erhält die Fuktion den Spielzustand „gameSpielzustand“, zu dem die Entscheidung über den nächsten Zug getroffen werden soll. In Zeile 2 wird nun überprüft, ob die Verwendung der Starttabellen aktiviert ist und zusätzlich in dem aktuellen Spiel weniger als 21 Züge gespielt wurden. Die zweite Komponente der „if“-Abfrage erfolgt, da die Startbibliothek maximal Strategien bis zum 20-ten Zug enthält.
Ist dies der Fall, so werden in Zeile 3 alle gemäß der Startbibliothek in Frage kommenden Züge ermittelt. Steht mindestens ein Zug zur Auswahl (Z. 4), so wird dieser gespielt (Z. 5). Nun werden die im vorherigen Zug ermittelten Wahrscheinlichekiten gelöscht(Z. 6) und das Symbol, welches zur Repräsentation des eigenen Spielers verwendet wird zwischengespeichert (Z.6).
Ist die Option zur Verwendung eines Präprozessors aktiviert, wird die Abfrage in Zeile 8 positiv ausgewertet. In diesem Fall wird der selektierte Präprozessor aufgerufen (Z. 9). Danach wird ermittelt ob die Option zur verwendung mehrer Prozesse aktiviert ist (Z. 10). Ist dies nicht der Fall, so werden „big_n“ zufällige Spiele gespielt (Z. 11). Die dazu aufgerufene Hilfsfunktion gibt ein Object des Typs „Dictionary“ zurück. Dieses enthält für jeden möglichen Zug die Anzahl der gewonnen Spiele wenn dieser Zug zuerst gespielt wurde und die Gesamtanzahl von Spielen.
Ist die Option zur Verwendung mehrere Prozesse aktiviert, so wird der „else“-Zweig in den Zeilen 12 bis 20 ausgeführt:
Dazu wird zunächst die Anzahl der verfügbaren Prozessorkerne ermittelt (Z. 13). Im weiteren

Verlauf werden entsprechend viele Prozesse verwendet. So können alle verfügbaren Prozessorkerne genutzt werden, ohne einen zu großen Verwaltungsaufwand für die Prozesse zu generieren. In Zeile 14 wird dann ein entsprechender Pool von Prozessen angelegt und durch die Anweisung der Zeilen 15 und 16 spezifiziert, dass mittels dieser Prozesse so häufig wie Prozesse vorhanden sind asynchron jeweils „`big_n`“ geteilt durch die Anzahl der Prozesse zufällige Spiele simuliert werden. Da die entsprechende Funktion eine Ganzzahl erwartet, kommt an dieser Stelle die Ganzzahldivison „`//`“ zum Einsatz. Dadurch ergibt die Summe aller durchgeführten Simulationen ggf. nicht „`big_n`“; bei entsprechend großen Werten kann dies jedoch vernachlässigt werden.

Da die Spiele asynchron simuliert werden, muss die Berechnung zunächst durch Aufruf der Funktion „`get`“ an jedem Listenelement angestoßen werden. Für das erste Element geschieht dies in Zeile 17. Für die weiteren Elemente erfolgt dies in der Schleife zur Zusammenführung der Ergebnisse. In Zeile 20 werden dann alle Prozesse wieder geschlossen.

Nun muss anhand der Daten zu gewonnenen und insgesamt gespielten Spiele für jeden Zug die Gewinnwahrscheinlichkeit berechnet werden. Dies erfolgt in den Zeilen 21 bis 23 indem der Quotient aus den beiden Werten gebildet wird (Z. 23)

Nun wird jener Zug mit der größten Gewinnwahrscheinlichkeit ermittelt (Z. 24) und and die aufrufende Funktion zurückgegeben (Z. 25).

Quelltext 4.10: get_move Funktion des Monte-Carlo Agenten

```
1  def get_move(self, game_state: Othello):
2      if self.use_start_lib and game_state.get_turn_nr() < 21:
3          moves = self.start_tables.get_available_moves_of_start_tables(game_state)
4          if len(moves) > 0:
5              return UtilMethods.translate_move_to_pair(moves[random.randrange(len(moves))])
6      self.move_probability.clear()
7      own_symbol = game_state.get_current_player()
8      if self.preprocessor is not None:
9          self.preprocessor(game_state, self.preprocessor_parameter, self.heuristic)
10     if not self.use_multiprocessing:
11         winning_statistics = MonteCarlo.play_n_random_games(own_symbol, game_state, self.big_n)
12     else:
13         number_of_processes = mp.cpu_count()
14         pool = mp.Pool(processes=number_of_processes)
15         list_of_stats = [pool.apply_async(MonteCarlo.play_n_random_games, args=(own_symbol,
16                 game_state.deepcopy(), self.big_n // number_of_processes)) for _ in range(number_of_proce
17         winning_statistics = list_of_stats[0].get()
18         for single_list in list_of_stats[1:]:
```

```
            MonteCarlo.combine_statistic_dicts(winning_statistics, single_list.get())
        pool.close()
    for single_move in winning_statistics:
        (games_won, times_played) = winning_statistics[single_move]
        self.move_probability[single_move] = games_won / times_played
    selected_move = max(self.move_probability.items(), key=operator.itemgetter(1))[0]
    return selected_move
```

Nutzung der gewichteten Random Funktion

Die gewichtete Zufalls Funktion bevorzugt bei der Auswahl eines Zuges Züge mit einer höheren Gewinnwahrscheinlichkeit des aktuellen Spielers. Die Funktion ist in Listing 4.11 abgebildet.
Der Funktion werden die aktuell verfügbaren Zugmöglichkeiten („`possible_moves`“), die Zugnummer
(„`turn_nr`“) und das aktuelle Spielersymbol („`own_symbol`“) übergeben (Z. 1).
Für jeden verfügbaren Zug wird die Gewinnwahrscheinlichkeit des Spielers bei Durchführung dieses Zuges ermittelt (Z.3f.) und aufsummiert (Z. 2 - 4). Die Gewinnwahrscheinlichkeit eines Zuges wird durch die Verwendung der in Kapitel 4.2.2 erläuterten Datenbank ermittelt (siehe Datenbank).
Nach der Summenbildung der Gewinnwahrscheinlichkeiten („`prob_sum`“) wird eine zufällige Zahl im Intervall [0,„`prob_sum`“] ermittelt (Z. 5).
Jedem Zug wird ein Intervall, welches die Größe der Gewinnwahrscheinlichkeit des Zuges entspricht, innerhalb des Intervalls [0,„`prob_sum`“] zugeordnet. Die Intervalle werden ab 0 beginnend aneinander angeordnet.
Durch die Ermittlung des Intervalls der zufällig ausgewählten Zahl wird diese einem Zug zugeordnet (Z. 8 - 13). Dieser Zug wird an die übergeordnete Funktion zurückgegeben.
Es gelten folgende mathematische Zusammenhänge der Zugauswahl des Spielers `player` eines bestimmten Zuges (`move`) aus den verfügbaren Zügen (`available_moves`):

- $\texttt{move} \in \texttt{available\_moves}; \texttt{available\_moves}$ ist eine Liste
- $\texttt{db\_prob}(\texttt{move}, \texttt{turn\_nr}, \texttt{player}) = P(\texttt{win} \mid \texttt{move} \land \texttt{turn\_nr} \land \texttt{player})$: Gewinnwahrscheinlichkeit des Spielers bei der Durchführung des Zuges `move` in Zugnummer `turn_nr`. Die Wahrscheinlichkeit wird aus der Datenbank ermittelt:
  $P(\texttt{win} \mid \texttt{move} \land \texttt{turn\_nr} \land \texttt{player}) = \frac{\texttt{won\_games}(\texttt{player},\texttt{move},\texttt{turn\_nr})}{\texttt{total\_played\_games}(\texttt{move},\texttt{turn\_nr})}$

- $\texttt{prob\_sum}(\texttt{available\_moves}, \texttt{turn\_nr}, \texttt{player}) = \sum_{\texttt{move} \in \texttt{available\_moves}} \texttt{db\_prob}(\texttt{move}, \texttt{turn\_nr}, \texttt{player}$
- Auswahlwahrscheinlichkeit des Zuges `move`:
  $P(\texttt{move} \mid \texttt{available\_moves} \wedge \texttt{turn\_nr} \wedge \texttt{player}) = \frac{\texttt{db\_prob}(\texttt{move}, \texttt{turn\_nr}, \texttt{player})}{\texttt{prob\_sum}(\texttt{available\_moves}, \texttt{turn\_nr}, \texttt{player})}$
- Zuordnung der zufällig ausgewählten Zahl zu einem Zug:
  $\texttt{map\_move} : [0, \texttt{prob\_sum}] \rightarrow \texttt{available\_moves}$
  In der folgenden Formel werden folgende Änderungen zur Steigerung der Verständlichkeit getroffen:
  - `a_m` ist die abkürzende Schreibweise für `available_moves`
  - Die Parameter `turn_nr` und `player` werden in der Funktion `db_prob` nicht angegeben, da diese in dieser Funktion konstante Parameter sind.

  $\texttt{map\_move}(\texttt{random\_nr}) = \{\texttt{a\_m}[i] \in \texttt{a\_m} \mid i \in \{0, .., len(\texttt{a\_m}) - 1\} \wedge (\sum_{k=0}^{i-1} \texttt{db\_proba\_m}[k] < \texttt{random\_nr}$
  $\wedge \sum_{k=0}^{i} \texttt{db\_proba\_m}[k] >= \texttt{random\_nr}\}$, wobei $\texttt{random\_nr} \in [0, \texttt{prob\_sum}]$
  Die Menge enthält immer nur ein Element, welches zurückgegeben wird, da die zweite Bedingung alle Züge unter der zufälligen Zahl herausfiltert in Verbindung mit der dritten Bedingung alle Züge über dem Intervall herausfiltert.

Durch die zufällige Auswahl eines Zuges aus der Summe `prob_sum` werden Züge mit einer hohen Gewinnwahrscheinlichkeit bevorteilt, aber Züge mit einer geringen Gewinnwahrscheinlichkeit werden zwar seltener, aber dennoch ausgewählt.

Quelltext 4.11: get_weighted_random Funktion des Monte-Carlo Agenten

```python
def get_weighted_random(possible_moves, turn_nr, own_symbol):
    prob_sum = 0.0
    for move in possible_moves:
        prob_sum += MonteCarlo._ml_database.get_likelihood(move, turn_nr, own_symbol)
    chose = random.uniform(0.0, prob_sum)
    move_nr = 0
    prob_sum = 0.0
    for move in possible_moves:
        move_nr += 1
        prob_sum += MonteCarlo._ml_database.get_likelihood(move, turn_nr, own_symbol)
        if prob_sum >= chose:
            return move
    return possible_moves[-1]
```

Nun wird das Verfahren an dem in Abbildung 4.4 abgebildeten Beispiels gezeigt. Im Beispiel existieren folgende Zugmöglichkeiten: „`c4`", „`e6`", „`f5`", „`d3`".
Im ersten Schritt werden die Gewinnwahrscheinlichkeiten des aktuellen Spielers für die jeweiligen Züge in der aktuellen Zugnummer ermittelt. Dies sind die Ergebnisse (siehe (1)):

- „`c4`": 40%
- „`e6`": 70%
- „`f5`": 55%
- „`d3`": 45%

Die Werte im Beispiel dienen der Demonstration und sind nicht aus der Datenbank entnommen.
Nun werden den Zügen Intervalle zugeordnet, welche die Länge der jeweiligen Wahrscheinlichkeit besitzen. Für den Zug „`c4`" beträgt die Länge beispielsweise „0,4".
Diese Intervalle werden aneinander angeordnet in der Reihenfolge der Zugmöglichkeiten aufsummiert (siehe (2)). Die Länge des Gesamtintervalls beträgt „2,1" („2,1" = 40% + 70% + 55% + 45%).
Anschließend wird aus diesem geschlossenen Intervall zufällig eine Zahl ausgewählt. In dem Beispiel wurde zufällig die Zahl „1,3" ausgewählt (siehe (3)).
Dieser Zahl (hier „1,3") wird ein Intervall zugeordnet. Im Beispiel liegt „1,3" im Intervall des Zuges „`f5`". Dadurch wird „`f5`" als ausgewählter Zug zurückgegeben (siehe (4)).

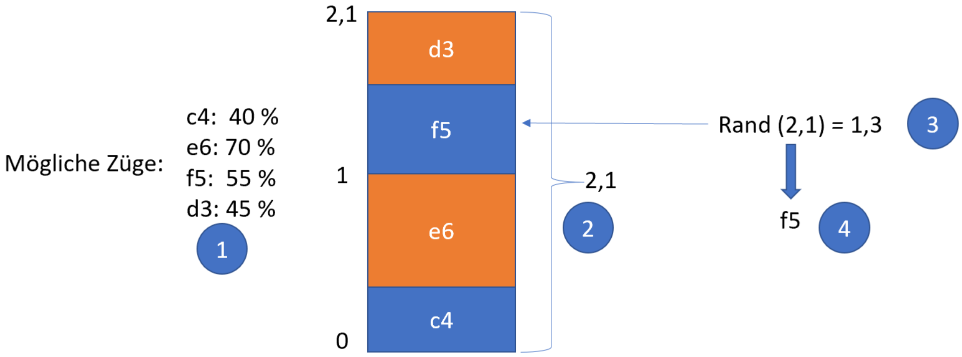


Abbildung 4.4: Auswahl eines gewichteten zufälligen Zuges

4.4.4 Alpha-Beta Pruning

Der Agent „Alpha-Beta Pruning“ basiert auf dem in Kapitel 3.2.2 beschriebenen Prinzip des Alpha-Beta Abschneidens. Wie bereits erläutert handelt es sich dabei um eine Tiefensuche, bei der weniger vielversprechende Zweige nicht bis zum Ende ausgewertet werden.
Wie in 3.2.3 erläutert wird dabei ab einer gewissen Suchtiefe eine Heuristik zur Bewertung eines Spielzustandes herangezogen. Dadurch, muss nicht der komplette Baum bis zum Ende ausgewertet werden.
Nachfolgend wird die in der Datei „`alphaBetaPruning.py`“ bedindliche Implementierung besprochen.

Parameter

Auch das detaillierte Verhalten des Agenten „Alpha-Beta Pruning“ kann durch verschiedene Parameter beeinflusst werden. Sie lauten:

- „`_use_start_libs`“: Bool’scher Wert der angibt ob Startbibliotheken verwendet werden sollen.
- „`_search_depth`“: Gibt die Tiefe an, ab der einzelne Zweige nicht weiter verfolgt werden und stattdessen die Heuristik zur Zustandsbewertung verwendet wird.
- „`_heuristic`“: Eine der in Kapitel 3.2.3 beschriebenen Heuristiken.
- „`_use_monte_carlo`“: Bool’scher Wert der angibt ob in der Tiefe „`search_depth`“ + 1 statt der Heuristik das Prinzip des Monte Carlo Agentens zur Bewertung der Züge verwendet werden soll.
- „`_mc_count`“: Anzahl der simuliert Spiele wenn das Monte Carlo Prinzip zur Bewertung des Zustandes verwendet wird.

Die Funktion „`value`“

Die in Listing 4.12 abgedruckte Funktion „`value`“ entspricht im wesentlichen der in Listing 3.1 abgedruckten und in 3.2.2 besprochenen „value“-Funktion des Alpha-Beta Abschneiden Algorithmus.

Hier soll einzig auf den sich unterscheidenden Basisfall (Zeile 3-6) eingegangen werden. Ist ein Endzustand erreicht, so ist das Spiel beendet und die Abfrage in Zeile 3 wird positiv ausgewertet. In diesem Fall wird der Wert der „`utility`“-Funktion für den aktuellen Spieler zurück gegeben (Z. 4). Um das Gewinnen bzw. Verlieren jedoch deutlich stärker zu gewichten als die Bewertungen der Heuristik, wird dies zuvor mit 1000 multipliziert.
Ist die maximale Suchtiefe erreicht, so wird die Abfrage in Zeile 5 positiv ausgewertet. In diesem Fall wird die Heuristik zur Zustandsbewertung herangezogen (Z. 6)

Quelltext 4.12: „`value`“ Funktion des Alpha-Beta Spielers

```
1  @staticmethod
2  def value(game_state: Othello, depth, heuristic, alpha=-1, beta=1):
3      if game_state.game_is_over():
4          return game_state.utility(game_state.get_current_player()) * 1000
5      if depth == 0:
6          return heuristic(game_state.get_current_player(), game_state)
7      val = alpha
8      for move in game_state.get_available_moves():
9          next_state = game_state.deepcopy()
10         next_state.play_position(move)
11         val = max({val, -1 * AlphaBetaPruning.value(next_state, depth - 1, heuristic, -beta, -alpha)})
12         if val >= beta:
13             return val
14         alpha = max({val, alpha})
15     return val
```

Änderungen an „`value_monte_carlo`“ Wird statt der Heuristik zur Bewertung der „Monte-Carlo Agent“ herangezogen, so wird statt der Funktion „`value`“ die angepasste Funktion „`value_monte_carlo`“ aufgerufen. Dabei wird in dem in Listing 4.12 in Zeile 6 beschriebenen „`if`“-Zweig eine Bewertung aller möglichen Züge vorgenommen und die Gewinnwahrscheinlichkeit des besten Zuges zurückgegeben.

Die Funktion „`get_move`“

Eine Funktion „`get_move`“ wird von allen Agenten bereitgestellt und in der „`main-game.py`“ zur Auswahl eines Zuges gerufen. Beschrieben wird der Zug durch ein Paar, welches die Koordinaten auf dem Spielbrett darstellt.

Nachfolgend wird die in Listing 4.13 abgedruckte Funktion diskutiert:
Die Übergabeparameter, sowie die Zeilen bis einschließlich 5 sind identisch zu der Implmentierung der im Kapitel 4.4.3 diskutierten „`get_move`“-Funktion des „Monte-Carlo“-Agenten. anders
In Zeile 6 wird ein Wörterbuch zur Speicherung der Bewertungen der Züge gemäß der „`value`“-Funktion angelegt. In diesem wird später die Bewertung als Schlüssel und als dazugehörigen Wert die Liste aller Züge die diese Bewertung erhalten haben gespeichert werden. Diese Datenstruktur wird durch die Schleife in den Zeilen 7 bis einschließlich 17 aufgbaut in dem über alle legalen Züge iteriert wird:
In den Zeilen 8 bzw. 9 wird der Ausgangszustand kopiert und der in der aktuellen Schleifeniteration betrachtete Zug „`move`“ ausgeführt. Nun wird die Bewertung mittels der „`value`“-Funktion vorgenommen. Dabei wird unterschieden ob gemäß des Parameters „`_use_monte_carlo`“ in der Tiefe „`_search_depth`“ mittels der Funktion „`value_monte_carlo`“ das Monte-Carlo Prinzip zur Bewertung herangezogen werden soll (Z. 10f) oder ob mittels der Funktion „`value`“ die Heuristik verwendet wird (Z. 13f).
Nun wird überprüft ob bereits Spielzüge mit dieser Bewertung in „`best_moves`“ gespeichert sind (Z. 15). Ist dies nicht der Fall, so wird die Bewertung mit der leeren Liste initialisiert (Z. 16). Anschließend wird der gerade betrachtete Zug „`move`“ an die entsprechende Liste angehängt.
Sobald alle legalen Züge betrachtet wurden, wird die maximale Bewertung ermittelt (Z. 18) und ein zufälliger Zug mit dieser Bewertung zurückgegeben (Z. 19).

Quelltext 4.13: get_move Funktion des Alpha-Beta Spielers

```
1   def get_move(self, game_state: Othello):
2       if self._use_start_lib and game_state.get_turn_nr() < 21:
3           moves = self._start_tables.get_available_moves_of_start_tables(game_state)
4           if len(moves) > 0:
5               return UtilMethods.translate_move_to_pair(moves[random.randrange(len(moves))])
6       best_moves = dict()
7       for move in game_state.get_available_moves():
8           next_state = game_state.deepcopy()
9           next_state.play_position(move)
10          if self._use_monte_carlo:
11              result = -AlphaBetaPruning.value_monte_carlo(next_state, self._search_depth - 1, self._heur
12                                                           mc_count=self._mc_count)
13          else:
14              result = -AlphaBetaPruning.value(next_state, self._search_depth - 1, self._heuristic)
15          if result not in best_moves.keys():
16              best_moves[result] = []
```

```
            best_moves[result].append(move)
        best_move = max(best_moves.keys())
        return best_moves[best_move][random.randrange(len(best_moves[best_move]))]
```

5 Evaluierung

Ziel dieses Kapitel ist es die im vorherigen Kapitel vorgestellte Implementierung zu testen. Dabei ergeben sich im wesentlichen zwei Testaspekte: Zum einen die vorgestellten Agenten und zum anderen die Bedeutung der in Kapitel 4.2.2 eingeführten Feldkategorien.

Die Evaluierung der einzelnen Agenten erfolgt dabei im wesentlichen dadurch die Agenten gegeneinander spielen zu lassen. Mittels einer genügend großen Anzahl von Spielen lässt sich darart ermitteln welcher der jeweils verwendeten Agenten im Mittel die meisten Spiele gewinnt und damit besser ist.
Die Bedeutung der Feldkategorien wird Anhand der Gewinnwahrscheinlichkeit einer Spielkategorie über den Spielfortschritt untersucht.

5.1 Evaluierung der einzelnen Agenten

Nun gilt es die in Kapitel 4 beschriebenen Agenten zu evaluieren. In Ermangelung eines Othello-Spielers, der über ein entsprechend gute Spielfähigkeiten verfügt, so dass er zu einem aussagekräftigen Vergleich herangezogen werden könnte, werden die einzelnen Agenten untereinander verglichen. Auf die gewählten Werte der Parameter soll hier nicht näher eingegangen werden. Die letzendlich verwendeten Parameter und das Verfahren um diese zu bestimmen wird in den Anschnitten Parameter des „Monte Carlo“-Agenten bzw. Parameter des „Alpha-Beta Pruning“-Agenten besprochen.
Bei der Bestimmung der Parameter hat sich gezeigt, dass die beim Agenten „Alpha-Beta Pruning“ gewählte Heuristik die Gewinnchance des Agenten wesentlich beeinflusst. Aus diesem Grund wird der „Alpha-Beta Pruning“-Agent mit allen zur Verfügung stehenden Heuristiken seperat betrachtet. Die sich daraus Ergebenden Vergleiche (Vgl.) sind in Tabelle 5.1 dargestellt. Der Agent „Alpha-Beta Pruning“ wird dabei durch „AB“ abgekürzt.

Vgl.	Agent 1	Agent 2
0	Random	Random
1	Monte Carlo	Random
2	AB (Nijssen 2008 Heuristik)	Random
3	AB (Stored Monte Carlo Heuristik)	Random
4	AB (Cowthello Heuristik)	Random
5	AB (mit anschließendem Monte Carlo)	Random
6	Random	Monte Carlo
7	Random	AB (Nijssen 2008 Heuristik)
8	Random	AB (Stored Monte Carlo Heuristik)
9	Random	AB (Cowthello Heuristik)
10	Random	AB (mit anschließendem Monte Carlo)
11	AB (beste)	Monte Carlo
12	Monte Carlo	AB (beste)

Tabelle 5.1: Durchgeführte Vergleiche

Vergleich mit dem Agent „Random“ Grundsätzlich gilt, dass der durch einen Agenten zur Auswahl von Spielzügen benötigte Rechenaufwand nur dann gerechtfertigt ist, wenn sich daraus in irgendeiner Form ein Vorteil ergibt, der Agent also besser spielt als ein Agent der beliebige Spielzüge durchführt. Aus diesem Grund werden alle Agenten zunächst mit dem „Random“-Agenten verglichen. Daraus ergeben sich zunächst die Vergleiche 1 bis 5.
Da ein Agent möglicherweise über bessere Gewinnchancen verfügt, wenn er als ein bestimmter Spieler auftritt gilt es außerdem zu Prüfen wie sich die Agenten verhalten, wenn sie auf der anderen Position verwendet werden. Daraus ergeben sich die Vergleiche 6 bis 10.

Vergleich der Agenten „Alpha-Beta Pruning“ und „Monte Carlo“ Bisher wurden die Agenten „Alpha-Beta Pruning“ und „Monte Carlo“ lediglich mit dem „Random“-Agenten verglichen. Interessant ist daher auch die Fragestellung wie die beiden Agenten gegeneinander Spielen. Hier sei vorweggegriffen, dass Anhand der im Abschnitt Ergebnisse der Vergleiche dargestellten Ergebnisse der Vergleiche 2 bis 5 bzw. 7 bis 10 festgestellt werden konnte, dass der Agent „Alpha-Beta Pruning“ mit der Heuristik „Cowthello“ die besten Ergebnisse erzielt. Daher wird diese für den Vergleich der Agenten untereinander herangezogen. Da aus den oben beschriebenen Gründen beide Varienten dieser Paarung betrachtet werden ergeben sich daraus die beiden Vergleiche 11 und 12.

Ergebnisse der Vergleiche Um zum einen ein möglichst aussagekräftiges Ergebnis zu kommen gilt es eine ausreichend große Anzahl von Spielen durchzuführen. Gleichzeitig steigt jedoch mit jedem weiteren Spiel die erforderliche Rechenzeit. Bei dem Versuch die Ziele von einer möglichst großen Spielzahl bei einer möglichst geringen Rechenzeit zu erreichen wurde mit verschiedenen Werten experimentiert. Zunächst wurden je Paarung 100 Spiele simuliert. Schnell stellte sich jedoch heraus, dass sich die Gewinnwahrscheinlichkeit für einzelne Agenten bei der Durchführung von 200 Spielen stark unterscheidet. Aus diesem Grund wurde für die Mehrzahl der in Tabelle 5.1 angegebenen Paarungen stattdessen 1000 Spiele durchgeführt. Bei einigen Kombinationen war es aus Gründen der erforderlichen Rechenzeit jedoch nicht möglich 1000 Vergleiche durchzuführen. Daher zu jedem Vgl. die Anzahl der Durchgeführten Spiele, sowie jeweils die Gewinnwahrscheinlichkeit angegeben. Die Ergebnisse dieser Vergleiche finden sich in Tabelle 5.2:

Vgl.	Anz. Spiele	Gewonnene Spiele				∅ Rechenzeit je Spiel [s]	
		Agent 1		Agent 2		Agent 1	Agent 2
		Anz.	Anteil [%]	Anz.	Anteil [%]		
0	1000	429	42,9	527	52,7	0,00027	0,00026
1	1000	1000	100,0	0	0,0	118,21435	0,00235
2	1000	436	43,6	465	46,5	253,5394	0,00055
3	1000	458	45,8	454	45,4	271,86036	0,00031
4	1000	564	56,4	324	32,4	156,59706	0,00056
5	200	132	66,0	60	30,0	550,9348	0,00045
6	1000	0	0,0	1000	100,0	0,00235	120,86694
7	1000	452	45,2	542	54,2	0.00045	224.09328
8	1000	446	44,6	550	55,0	0,00034	234,01762
9	1000	289	28,9	709	70,9	0,00049	149,17649
10	200	33	16,5	166	83,0	0,00069	717,85712
11	7	0	0,0	7	100,0	309,24247	490,69960
12	-	-	-	-	-	-	-

Tabelle 5.2: Ergebnisse der Vergleiche

MCvsABP(MC)

5.2 Anpassung der Parameter der verschiedenen Agenten

Der Wahl der Parameter lag die Prämisse zugrunde, dass die Gesamtberechnungsdauer eines Agenten pro Spiel etwa fünf Minuten betragen sollte. Zur Ermittlung der Werte wurde ein Mircosoft® Surface Pro™ der 6. Generation mit einer CPU der Intel® Core™ i7 Familie mit 2,11 GHz Taktrate, als Referenzgerät festgelegt. Dementsprechend wurden die Parameter der einzelnen Agenten so angepasst, dass die Berechnungsdauer im Mittel unterhalb dieser Marke liegt. Einzelne Spielabläufe können sich jedoch sehr stark unterscheiden. Entsprechend können sich bei einzelnen Spielen auch größere Abweichungen ergeben. Dabei gibt es sehr schnelle Spiele, bei denen bspw. ein Spieler gewinnt ohne dass das komplette Spielfeld besetzt ist, aber auch sehr lange Spiele, bei denen für jeden Zug viele mögliche Folgezüge evaluiert werden müssen.
Da die Testsläufe über eine sehr lange Laufzeit verfügen wurden die in Tabelle 5.2 angegebenen Tests nicht auf dem zur Ermittlung der Parameter herangezogenen Mircosoft® Surface Pro™ ermittelt. Zum Einsatz kamen stattdessen virtualisierte Server die einer Intel® Xeon® Broadwell E5-2680 v4 CPU mit einer Taktrate von 2.4 GHz aufbauen. Da die Verfasser dieser Arbeit die Server für andere Projekte bereits angemietet haben, jedoch nicht auslasten, entstehen ihnen so durch die Berechnung der Vergleiche keine zusätzlichen Energiekosten. Darüber hinaus ist es möglich das Mircosoft® Surface Pro™ anderweitig zu verwenden.
Nachfolgend werden die Ergebnisse der Anpassung der Parameter gemäß dem oben Beschriebenen Verfahren diskutiert.

5.2.1 Parameter des „Monte Carlo“-Agenten

Die beim Monte Carlo Agenten gemäß Kapitel 4.4.3 zur Verfügung stehenden Parameter werden wie folgt festgesetzt:

- „`big_n`“: 2000
- „`use_start_libs`“: `True`
- „`preprocessor`“: `None`
- „`preprocessor_parameter`“: `None`
- „`heuristic`“: `None`
- „`use_multiprocessing`“: `True`

Der Parameter „`big_n`“ wurde, nachdem alle übrigen Parameter festgesetzt wurden empirisch ermittelt, indem mehrere Spiele mit unterschiedlichen „`big_n`“ gespielt wurden und die benötigte Spielzeit mit dem Zielwert von fünf Minuten verglichen wurde. Aus dieser Testreihe ergab sich einen „`big_n`“ Wert von 2000. Die Gesamtdauer der Spiele beträgt dabei meist drei bis fünf Minuten. Wird der Parameter weiter erhöht, wird die Gesamtspielzeit des Agenten von fünf Minuten überschritten.
Das setzen des Wertes„`use_start_libs`“ auf „`True`“ verkürzt die Berechnungsdauer der ersten Züge enorm, da die Züge aus der Starttabellen gelesen werden und nicht berechnet werden. Dadurch steht im Anschluss mehr Zeit für die Simulation von Spielen zur Verfügung, ohne dass dies dabei einen negativen Effekt auf das Zeitlimit von fünf Minuten hat. Zwar können die Felder, welche durch in der Eröffnungsphase gesetzt werden im weiteren Spiel noch mehrfach gedreht werden, andererseits sind keine willkürlichen Züge in der Tabelle gespeichert, sondern bewährte Eröffnungszüge.
Der Parameter „`preprocessor`“ wird auf „None“ gesetzt. Damit wird der Präprozessor deaktiviert. Durch die Nutzung eines Präprozessors wird die Anzahl der Zugmöglichkeiten je Zug verkleinert. Dadurch können innerhalb des Zeitlimits von fünf Minuten theoretisch ebenfalls mehr Spiele durchgeführt werden. Jedoch muss zur Verwendung des Präprozessors eine Heuristik verwendet werden, die ebenfalls berechnet werden muss. Da sich der beschriebene Vorteil damit ausglicht wird im Sinne eines einfacheren Algorithmus aud den Präprozessor verzichtet.
Durch die Deaktivierung des Präprozessors, werden die Parameter
„`preprocessor_parameter`“ und „`heuristic`“, da diese nur im Prepozessor verwendet werden, überhaupt nicht initialisiert. Damit stehen sie auf dem Standardwert für nicht initialisierte Paramter: „`None`“.
Der Parameter „`use_multiprocessing`“ wird auf „`True`“ gesetzt und die Verwendeung mehrerer Prozessoren damit aktiviert. Durch das Aufteilen der zu simulierenden Spiele auf mehrere Prozessorkerne und die Aggregation der einzelnen Werte wird zwar eine gewisse Zeit beansprucht, der positive Effekt durch die Parallelisierung überwiegt jedoch ab circa 80 simulierten Spielen. Da für ein aussagekräftiges Ergebnis bereits mehr als 80 Spiele simuliert werden müssen, können durch die Parallelierung in fünf Minuten insgesamt deutlich mehr Spiele simuliert werden und damit kann wiederum eine bessere Abschätzungen über die Gewinnwahrscheinlichkeiten beim Spielen eines bestimmten Zuges erzielt werden.

5.2.2 Parameter des „Alpha-Beta Pruning“-Agenten

Das gleiche Vorgehen aus dem vorherigen Abschnitt wird nun ebenfalls auf den „Alpha-Beta Pruning“-Agenten angewendet. Dabei werden die Parameter wie folgt festgesetzt:

- „`_use_start_libs`“: `True`
- „`_search_depth`“: 5 bei Verwendung der Heuristiken, 2 bei Verwendung von Monte Carlo
- „`_heuristic`“: variiert je nach Vergleich
- „`_use_monte_carlo`“: variiert je nach Vergleich
- „`_mc_count`“: variiert je nach Vergleich

„`_use_start_libs`“ wird auf „`True`“ gesetzt, da so die Verwendung mehrerer Prozessorkerne aktiviert wird und damit die Ausführungszeit eines Spiels merklich veringert wird. Je nach Spielzug des Gegners können bis zu 21 Züge aus der Datenbank gelesen werden und müssen nicht berechnet werden.
„`_heuristic`“ wurde jeweils variiert, damit ein Vergleich der Heuristiken möglich ist. Bei der Verwendung einer Heuristik wurden die Optionen hinter den Parametern „`_use_monte_carlo`“ und „`_mc_count`“ deaktiviert „`_use_monte_carlo`“ wird auf dazu auf „False“ gesetzt. Die ermittelte Suchtiefe („`_search_depth`“) beträgt „5“. Bei höheren Suchtiefen wird das Berechnungszeitlimit von fünf Minuten überschritten.

Neben Heuristiken kann die Alpha-Beta Suche auch Monte Carlo zur Ermittlung des Spielzustandswertes nach der Alpha-Beta Suche genutzt werden.
„`_use_monte_carlo`“ wird dazu auf „True“ gesetzt, die Option also aktiviert aktiviert. „`_mc_count`“ gibt dann die Anzahl der in den Blattknoten durchgeführten Monte Carlo Spiele an. Dieser Parameter wurde in diesem Fall auf 10 gesetzt, da bereits die Alpha-Beta Suche ohne eine Bewertung des Zustands mit Monte Carlo relativ lange dauert. Deshalb können in den Blattknoten im Vergleich zum reinen Monte Carlo Agenten nicht annähernd so viele Spiele durchgeführt werden.
„`_search_depth`“ wurde durch empirisches Testen der Laufzeit auf „2“ festgelegt.

5.3 Evaluierung der Bedeutung verschiedener Feldkategorien

Im nachfolgenden Abschnitt wird anhand der für die „Stored Monte Carlo“-Heuristik erstellten Datenbank die Bedeutung verschiedener Feldkategorien im Spielverlauf betrachtet.
Dies wird Anhand von verschiedenen Diagrammen durchgeführt. Dazu wird zunächst die Methode zur Erstellung der Diagramme besprochen. Daran schließt sich die Präsentation der Diagramme an sich an. Schließlich wird der Abschnitt durch die Besprechung der Ergebnisse beendet.

5.3.1 Methode und Ergebnisse

Methode Für die Beurteilung der Bedeutung einzelner Feldkategorien wurde die mit der „Stored Monte Carlo Heuristik“ aufgebaute Datenbank verwendet. Diese wurde zur Durchführung der in Kapitel 5.1 durchgeführten Vergleiche aufgebaut.
Zu diesem Zweck wurden, wie in Kapitel 4.2 erläutert, mehrere Partien zwischen zwei „Random“-Agenten gespielt und die Ergebnisse gespeichert. Im vorliegenden Fall wurden 140.000 Spiele simuliert. Nun kann für jedenSpieler zu jeder Zugnummer und jede Feldkategorie die Gewinnwahrscheinlichkeit für das Spielen einer Feldkategorie berechnet werden als:

$$\frac{\texttt{Anzahl gewonnene Spiele}}{\texttt{Anzahl insgesamt gespielte Spiele}}.$$

Um eine Division durch 0 zu vermeiden, wird die berechnete Gewinnwahrscheinlichkeit auf 0 gesetzt sofern bei der jeweiligen Zugnummer nie eine derartige Feldkategorie gespielt wurde. Aufgrund der großen Anzahl von zufällig gespielten Spielen ist in diesem Fall davon auszugehen, dass es zu der jeweiligen Zugnummer keine Möglichkeit gibt eine derartige Feldkategorie zu spielen.
Insgesamt ergibt sich damit pro Spieler eine Relation $\{0, ..., 59\} \times \{0, ..., 8\} \mapsto [0, 1]$, die für die Zugnummer und für die Feldkategorie jeweils die Gewinnwahrscheinlichkeit angibt.

Ergebnisse Um die große Anzahl von $60\ \texttt{Zugnummern} * 9\ \texttt{Zugkategorien} = 540$ Wahrscheinlichkeitswerten pro Spieler übersichtlich darstellen zu können, werden diese grafisch aufbereitet. Da hierfür ausschließlich zweidimensionale Diagramme zum Einsatz kommen, sollen besteht die

Möglichkeit hier nach der Zugnummer oder der Feldkategorie zu schneiden. Die Abbildungen 7.1 bis 7.9 im Anhang geben die Gewinnwahrscheinlichkeiten für den Schnitt nach der Feldkategorie an.
Die Abbildungen 7.10 bis 7.22 im Anhang geben die Gewinnwahrscheinlichkeiten für den Schnitt nach der Zugnummer an. Da aus Platzgründen nicht alle 60 der sich daraus ergebenden Abbildungen abgedruckt werden sollen, wurde dabei die willkürlich gewählte Schrittweite von 5 Zugnummern gewählt. Die Situation für den Zug 59 wird trotzdem angegeben.

5.3.2 Besprechung der Ergebnisse

Bei der Betrachtung der Diagramme fallen einige Aspekte sofort auf:

1. **Spieler Zwei besitzt fast durchgehend eine höhere Gewinnwahrscheinlichkeit als Spieler 1:**
   Dies liegt vermutlich darin begründet, dass Spieler 1 das Spiel eröffnet. Fasst man zwei aufeinanderfolgende Züge nun als Halbzüge eines kompletten Spielzuges auf, so kann Spieler 2 immer direkt auf den Zug des Spieler 1 reagieren.

2. **Einige Feldkategorien spielen erst nach einer gewissen Zugnummer eine Rolle:**
   Dies liegt darin begründet, dass ein Spielstein nur neben ein bereits besetztes Feld gelegt werden darf. Zu Beginn des Spiels sind jedoch nur die vier Steine in der Mitte des Spielfeldes gesetzt. Entsprechend dauert es einige Züge bis gewisse Feldkategorien wie bspw. die Eckfelder überhaupt erreicht werden können.

3. **Die Gewinnwahrscheinlichkeit für einen einzelnen Spieler schwankt für die Zugnummern teilweise sehr stark:**
   Dabei fällt auf, dass jeweils die Wahrscheinlichkeiten bei geraden Zugnummern und die Wahrscheinlichkeiten bei ungeraden Zugnummern auf einem ungefähr gleichen Niveau liegen. Zwischen zwei aufeinanderfolgenden Zügen gibt es hingegen meist eine deutliche Schwankung.

6 Fazit

6.1 Bewertung der Agenten

In diesem Kapitel werden die Agenten „Monte Carlo“ und „Alpha-Beta Pruning“ verglichen.

6.1.1 Der Agent „Monte Carlo“

Zsf Add

Gegen den Agenten „Random“ Wie in Tabelle 5.2 anhand der Vergleiche 1 bzw. 6 deutlich wird, hat der „Monte Carlo“-Agent sämtliche der insgesamt 2000 Testspielen gegen den „Random“-Agenten gewonnen. Dabei ist bis auf eine geringe Abweichung der durchschnittlich benötigten Rechenzeit pro Spiel unerheblich ob der Agent als Spieler 1 oder Spieler 2 auftritt. Damit ist der mit dem Spiel des Agenten verbundene Rechenaufwand in so fern berechtigt, dass er bessere Ergebnisse liefert als ein Spieler der zufällige Züge durchführt.
Im Vergleich mit menschlichen Spielern ist jedoch davon auszugehen, dass diese keine zufälligen Züge ausführen. Stattdessen ist, ein entsprechendes Spielniveau vorausgesetzt, damit zu rechnen, dass ein Spieler Kenntnisse über das Spiel und durch Erfahrung verfeinerte Strategien einbringt. Dieser Vergleich konnte im Rahmen der Arbeit leider nicht durchgeführt werden und ist damit eine Fragestellung für weitere Untersuchungen.

Weitere Variationsmöglichkeiten In den zur Bestimmung der für den Agenten verwendeten Parameter konnte beobachtet werden, dass die Gewinnwahrscheinlichkeit des Agenten von der Anzahl der simulierten Spiele abhängt. In den betrachteten Fällen konnte durch die Erhöhung der Anzahl simulierter Spiele, also die Erhöhung des Parameters „`big_n`“, im allgemeinen eine Verbesserung der Gewinnwahrscheinlichkeit des Agenten erzielt werden. Gleichzeitig zeigte Nijssen 2007, dass sich dieser Effekt mit steigender Größe von „`big_n`“ abschwächt. Zusätzlich dazu steigt mit der Anzahl der durchgeführten Simulationen auch die benötigte Rechenzeit. Zu

Untersuchen wäre daher ob das überschreiten der Rechenzeit von fünf Minuten auf dem Referenzgerät eine weitere Verbesserung bringt und damit den erhöhten Aufwand rechtfertigt.

6.1.2 Der Agent „Alpha-Beta Pruning“

Zsf

add

Im Vergleich zum Agenten „Ramdom“ Im Vergleich mit dem zufällig spielenden Agenten gewann der Agent „Alpha-Beta Pruning“ in allen Fällen mehr Spiele. Deutlich wird auch, dass der Erfolg des Agenten im wesentlichen von der verwendeten Heuristik abhängt. Auf die einzelnen Heuristiken, wird im Abschnitt 6.2 genauer eingegangen.

Immer noch so?

An dieser Stelle ist festzuhalten, dass auch der „Alpha Beta Pruning“-Agent, die Wahl einer guten Heuristik vorausgesetzt, den Rechenaufwand in so fern rechtfertigt, das Spiel des Agenten besser ist als das des Zufällig agierenden. Wie oben ist auch hier die Gewinnwahrscheinlichkeit gegen menschliche Spieler Gegenstand weiterer Untersuchungen.
Auffällig ist außerdem, dass im Vergleich zu dem „Monte Carlo“-Agenten eine deutlich längere Rechenzeit benötigt wird um zu einem insgesamt schlechteren Ergebnis zu kommen. Auf die Performance der beiden Agenten im direkten Vergleich wird in Abschnitt 6.1.3 eingegangen.

MC

Weitere Variationsmöglichkeiten Gemäß der in Kapitel 3.2.2 beschriebenen Theorie hinter dem Agenten „Alpha-Beta Pruning“ kann durch eine größere Suchtiefe „`search_depth`“ die Genauigkeit mit der die Bewertung eines Zuges der realitätsgetreuen Bewertung entspricht erhöht werden. Da sich der Suchbaum mit jeder Ebene stärker verzweigt ist in diesem Fall jedoch mit einer exponentiell steigenden Rechenzeit zu rechnen. Mit der vorgestellten Implementierung wäre dies unter dem Zeitaspekt jedoch keine Oprion. Entsprechend stellt sich hier die Frage in wie weit der Algorithmus des Alpha Beta Abschneidens parallelisiert werden kann um alle verfügbaren Prozessoren eines Computersystems zu benutzen und damit innerhalb der fünf Minuten Rechenzeit die der Anwender bereit ist zu warten, den Suchbaum noch tiefer zu durchsuchen.

6.1.3 Vergleich der Agenten „Monte Carlo“ und „Alpha-Beta Pruning“

Der „Monte Carlo“ Agent spielt deutlich besser als der „Random“ Agent. Der „Alpha-Beta Pruning“ Agent ist bei der Verwendung der „Cowthello“ Heuristik ebenfalls deutlich besser als der „Random“ Agent, aber dennoch schlechter als der „Monte Carlo“ Agent bei ähnlicher Spieldauer.
Da der „Monte Carlo“ Agent in der Spieltheorie einfach nur aus einer gewissen Anzahl an zufälligen Spielen ab der aktuellen Spielposition den besten Folgezug zurückgibt, besitzt dieser Agent keinerlei Wissen über das Spielprinzip des Spiels Othello.
Dies sollte durch die Verwendung des „Alpha-Beta Pruning“ Agenten verbessert werden. Dieser basiert grundlegend auf einer angepassten MiniMax-Suche. Diese deckt den Aspekt des Spiels ab, dass gute Züge des eines Spielers gleichzeitig schlechte Züge des Gegners sind.
Der erwartete Effekt trat allerdings nicht ein. Dies kann mehrere Gründe haben:

- Die verfügbaren Heuristiken, welche in den Blattknoten der „Alpha-Beta Pruning“ Suche verwendet werden, sind zu ungenau. Beispielsweise wurde in Kapitel 6.2.2 festgestellt, dass die Heuristik „Stored Monte-Carlo“ ungeeignet ist. Durch die große Spanne der Gewinnwahrscheinlichkeiten des „Alpha-Beta Pruning“ Agenten je nach verwendeter Heuristik ist es denkbar, dass noch bessere Heuristiken existieren.

- Die „Alpha-Beta Pruning“ Suche benötigt sehr viel Zeit. Durch die große Verzweigung existieren sehr viele Blattknoten, für welche ein Wert ermittelt werden muss. Dies geschieht durch die Verwendung einer Heuristik oder der „Monte Carlo“ Suche.
  In der Zeit, welche der „Alpha-Beta Pruning“ Agent zur Suche benötigt, kann der „Monte Carlo“ Agent deutlich mehr zufällige Spiele berechnen. Dadurch wird der Vorteil des „Alpha-Beta Pruning“ Agenten durch die große Anzahl der durchgeführten Züge ausgeglichen.

- Die „Alpha-Beta Pruning“ Suche besitzt eine zu geringe Suchtiefe. Durch eine größere Suchtiefe werden bessere Ergebnisse erreicht. Allerdings steigt die Berechnungszeit eines Zuges selbst bei der Erhöhung der Suchtiefe nur um eins sehr stark an. Dadurch kann der Agent nur bis zu einer bestimmten Suche (max. 5) eine „Alpha-Beta Pruning“ Suche durchführen, bevor die Wartedauer auf den nächsten Zug den erwarteten Nutzen übersteigt.

6.2 Bewertung der Heuristiken

Max

6.2.1 Die Heuristik „Nijssen 07“

Max

6.2.2 Die Heuristik „Stored Monte-Carlo“

Der „Stored Monte-Carlo“ Agent gewinnt in beiden Spielkombinationen häufiger als der „Random“ Agent (siehe Abbildung 5.2 Vergleiche 3 und 8). Im Vergleich mit den anderen Heuristiken ist die Gewinnwahrscheinlichkeit sehr gering.
Im Spiel „AB (Stored Monte Carlo Heuristik) “ gegen „Random“ (siehe Vergleich 3) gewinnt der Agent in 1000 Spielen nur vier Spiele mehr als der zufällige Spieler (458 zu 454). In den 1000 Spielen „Random“ gegen „AB (Stored Monte Carlo Heuristik) “ gewinnt die Heuristik mit 446 zu 550 gewonnenen Spielen (siehe Vergleich 8).
Aus Vergleichen 3 und 8 lassen sich folgende Erkenntnisse ableiten:

- Der Agent „Alpha-Beta Pruning“ gewinnt bei der Verwendung der „Stored Monte-Carlo“ als zweiter Spieler deutlich mehr Spiele als als erster Spieler.

- Wird der Agent als erster Spieler eingesetzt ergibt sich keine deutlich höhere Gewinnwahrscheinlichkeit als die des zufälligen Spielers. Dies bedeutet, dass der Agent sehr schlecht spielt, also quasi zufällig Züge auswählt.

- Auffällig sind die 88 Spiele, welche im Vergleich unentschieden endeten. Im Gegensatz gab es im Vergleich 8 nur vier unentschiedene Spiele. Die Gewinnwahrscheinlichkeit des zufälligen Spielers bleibt allerdings annähernd gleich bei 45,4% bzw. 44,6%. Man könnte dieses Ergebnis so interpretieren, dass der Agent in Vergleich die Spiele zwar nicht gewinnen konnte, aber immerhin verhindern konnte, dass diese Spiele verloren wurden.

- Die Spiele im Vergleich 8 sind durchschnittlich 37,8 Sekunden schneller die Spiele im Vergleich 3.

Die Nutzung der Heuristik „Stored Monte-Carlo“ wird aus den o.g. Ergebnissen nicht empfohlen.
Es gibt mehrere mögliche Ursachen der schlechten Gewinnwahrscheinlichkeiten der Heuristik.

Die erste Ursache ist, dass die Datenbank, auf welcher die Heuristik basiert, nicht die Gewinnwahrscheinlichkeit des Spielers bei der Durchführung eines Zuges in der aktuellen Zugnummer im aktuellen Spiel liefert. Stattdessen gibt die Datenbank die Gewinnwahrscheinlichkeit des Spielers zurück, der in der aktuellen Zugnummer den Zug ausführt, zurück.
Der Unterschied zwischen den Aussagen besteht darin, dass die Datenbank den aktuellen Spielzustand (bereits durchgeführte Spielzüge) vernachlässigt und nur die statistische Wahrscheinlichkeit über alle Züge zurückgibt, in welchen in der aktuellen Zugnummer der Zug zu einem Gewinn geführt hat.
Dadurch können Spielsituationen auftreten, in welchen der Zug mit der höchsten Gewinnwahrscheinlichkeit der Datenbank schlechter ist als ein Zug mit einer geringeren Gewinnwahrscheinlichkeit, da die jeweilige Spielsituation die Wahrscheinlichkeiten stark beeinflusst.

Es müsste auch evaluiert werden, ob die Zusammenfassung der Spielfelder in zehn Spielkategorien eine zu starke Vereinfachung des Spielfeldes darstellt. Das Spielfeld ist zwar symmetrisch aufgebaut, es könnten aber dennoch Seiteneffekte auftreten.

Dies führt zu einer weiteren möglichen Ursache der schlechten Heuristik. Aus den in der Datenbank gespeicherten Tripel aus gewonnen Spielen des ersten / zweiten Spielers und der Gesamtanzahl der durchgeführten Spiele wird nur die Gewinnwahrscheinlichkeit berechnet. Die Gesamtanzahl der durchgeführten Spiele wird allerdings nicht berücksichtigt. Es kann durchaus vorkommen, dass einzelne Feldkategorien unterschiedlich oft gespielt werden. So kann eine Feldkategorie sehr selten gespielt werden, dann aber eine relativ hohe Gewinnwahrscheinlichkeit besitzen, und eine Feldkategorie sehr oft mit einer geringeren Gewinnwahrscheinlichkeit gespielt werden. Die Heuristik bevorzugt in diesem Fall die selten gespielte Feldkategorie, da die Gewinnwahrscheinlichkeit höher ist. Da die Gesamtanzahl aller durchgeführten Spielzüge einer Zugnummer konstant sind (140.000), ist es u.U. auch sinnvoll die Gesamtanzahl der Spiele einer Feldkategorie der verfügbaren Feldkategorien in die Berechnung der Heuristik einzubinden.

6.2.3 Die Heuristik „Cowthello“

Die „Cowthello“ Heuristik bietet die höchsten Gewinnwahrscheinlichkeiten der „Alpha-Beta Pruning“ Heuristiken. Folgende Ergebnisse können aus den Vergleichen 4 und 9 der Tabelle 5.2 abgeleitet werden:

- In den 1000 Spielen des „Alpha-Beta Pruning“ Agenten mit der „Cowthello“ Heuristik gegen den „Random“ Agenten gewinnt der „Alpha-Beta Pruning“ Agent 564 zu 324 Spiele.
- Die große Zahl von 112 unentschiedenen Spielen fällt bei dieser Heuristik ebenfalls auf.
- In den 1000 Spielen des „Random“ Agenten gegen den „Alpha-Beta Pruning“ Agenten mit der „Cowthello“ Heuristik gewinnt der „Alpha-Beta Pruning“ Agent 709 zu 289 Spiele.
- Die durchschnittliche Spielzeit des Agenten ist als zweiter Spieler ebenfalls geringer als die des Agenten als erster Spieler. Die Spielzeit der „Cowthello“ Heuristik ist die kleinste aller verwendeten Heuristiken.

6.3 Bewertung der Vorgehensweise

Outline: Max

6.4 Ausblick

Die in der „Stored Monte-Carlo“ Heuristik verwendetet Datenbank kann dazu genutzt werden, um einen verbesserten Agenten zu erstellen. Diese Funktionsweise des neuen Agenten ist der des „Monte Carlo“ Agenten mit verwendeter gewichteter Zufallsfunktion sehr ähnlich. Der Unterschied wäre das Aktualisieren der Datenbank nach der Durchführung jedes Spieles. Dadurch nähern sich die in der Datenbank gespeicherten Werte statistisch an die optimale Werte an.
Dieser Prozess ist aber zwangsläufig mit Schreibzuggriffen oder mindestens das Verändern des Datenbankenobjektes verbunden. Dies hat zwei schwerwiegende Auswirkungen:

- Die Parallelisierung der gleichzeitigen Ausführung zufälliger Spiele ist schwieriger, da nur ein Thread gleichzeitig in die Datenbank schreiben kann.

- Die zusätzliche Datenbankinteraktion benötigt mehr Zeit.

Das Resultat der beiden Auswirkungen ist, dass die Anzahl der durchgeführten zufälligen Spiele reduziert werden muss wenn die Berechnungszeit nicht wachsen soll.
Es ist zu evaluieren, ob der Mehrwert der Datenbanknutzung größer ist als der bisherige „Monte Carlo“ Agent, welcher eine höhere Anzahl an zufällig durchgeführter Spiele besitzt.

Bedeutung für die Methode

Notes

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

Berg, Matthias (o.D.). *Strategieführer*. http://berg.earthlingz.de/ocd/strategy2.php. [Online; accessed 20-January-2019].

Nijssen, J. A. M. (Juni 2007). *Playing Othello Using Monte Carlo*. https://project.dke.maastrichtuniversity.nl/games/files/bsc/Nijssen_BSc-paper.pdf.

o.V. (o.J). *Cowthello*. http://www.aurochs.org/games/cowthello/cowthello.js. [Online; accessed 25-03-2019].

o.V. (2005). *Reversi Openings*. http://samsoft.org.uk/reversi/openings.htm. [Online; accessed 24-03-2019].

o.V. (2015). *Spiele: Othello*. https://de.wikibooks.org/wiki/Spiele:_Othello. [Online; accessed 20-January-2019].

Ortiz, George and Berg, Matthias (o.D.). *Eröffnungsstrategie*. http://berg.earthlingz.de/ocd/strategy3.php. [Online; accessed 20-January-2019].

Russell, Stuart J. und Peter Norvig (2016). *Artificial intelligence: A modern approach*. Always learning. Pearson. ISBN: 978-0-13-604259-4.

Stroetmann, Prof. Dr. Karl (2019). „An Introduction to Arti

cial Intelligence - Lecture Notes in Progress“. In:

7 Anhang

Abb.: Bedeutung der Feldkategorien - Schnitt nach Feldkategorie

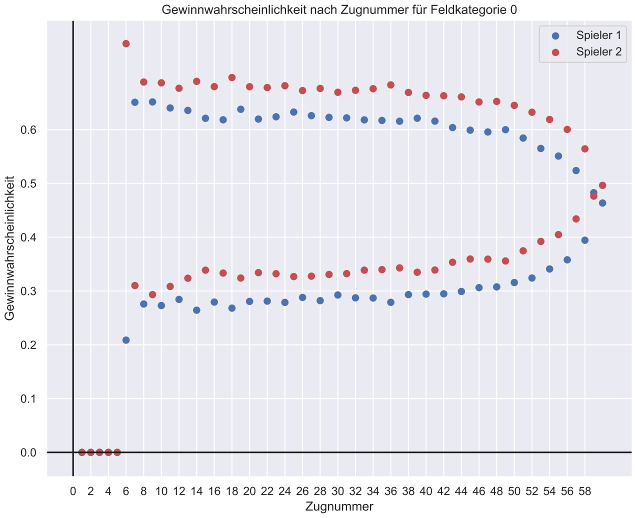


Abbildung 7.1: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 0

Abb.: Bedeutung der Feldkategorien - Schnitt nach Zugnummer

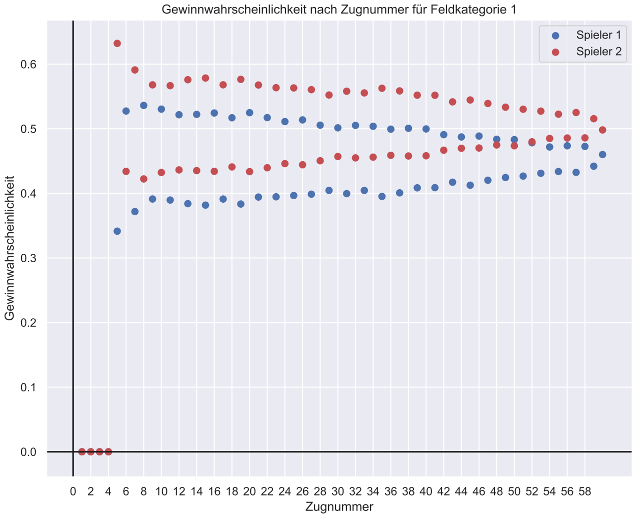


Abbildung 7.2: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 1

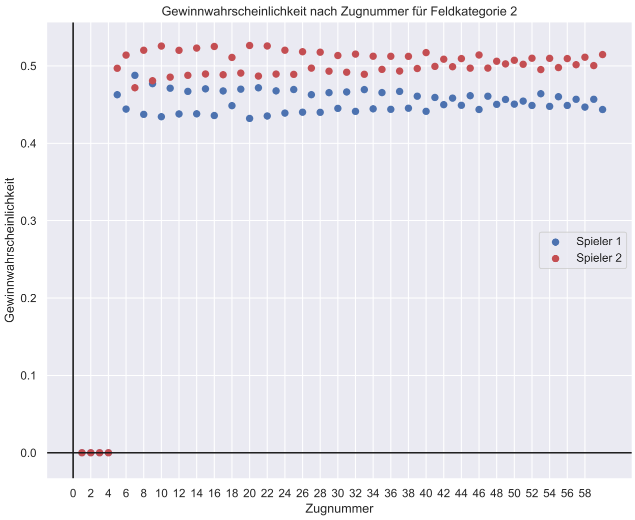


Abbildung 7.3: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 2

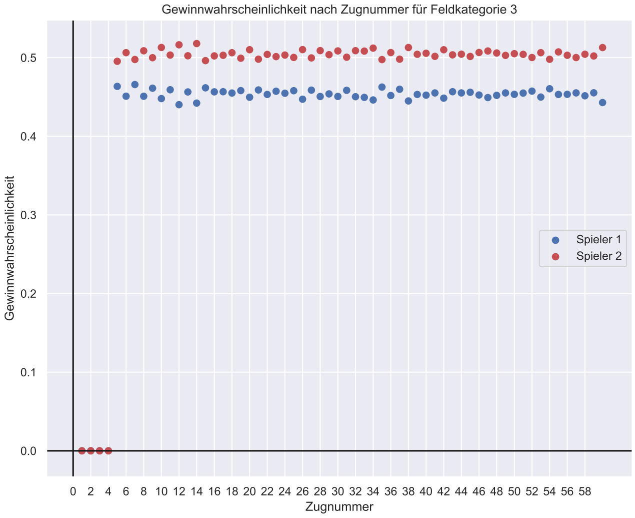


Abbildung 7.4: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 3

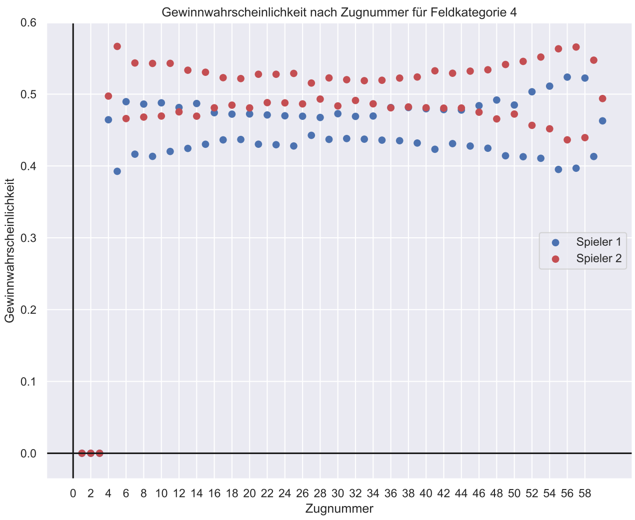


Abbildung 7.5: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 4

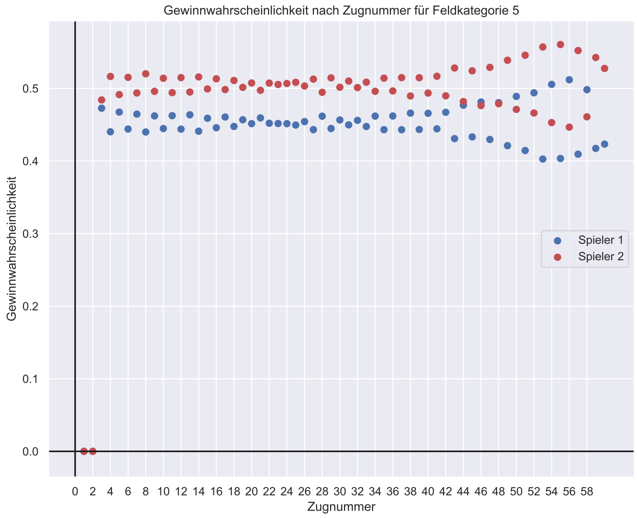


Abbildung 7.6: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 5

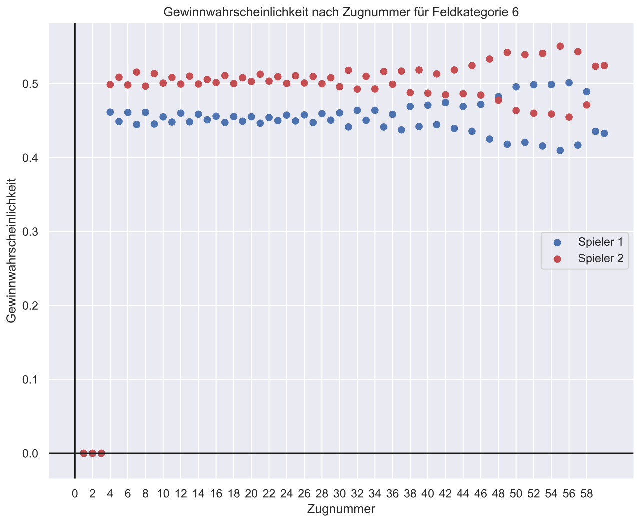


Abbildung 7.7: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 6

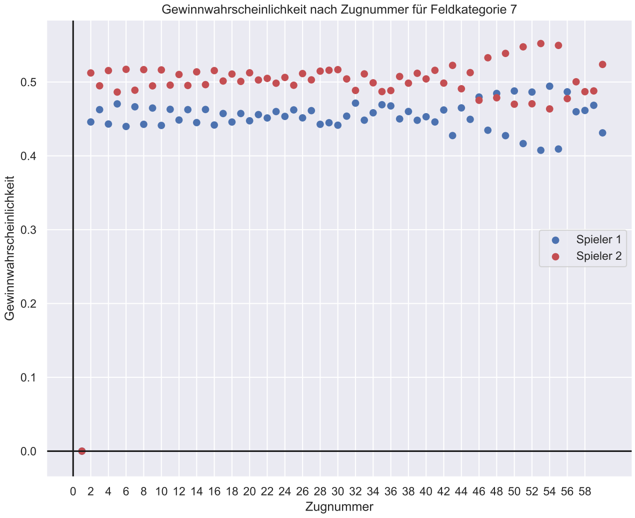


Abbildung 7.8: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 7

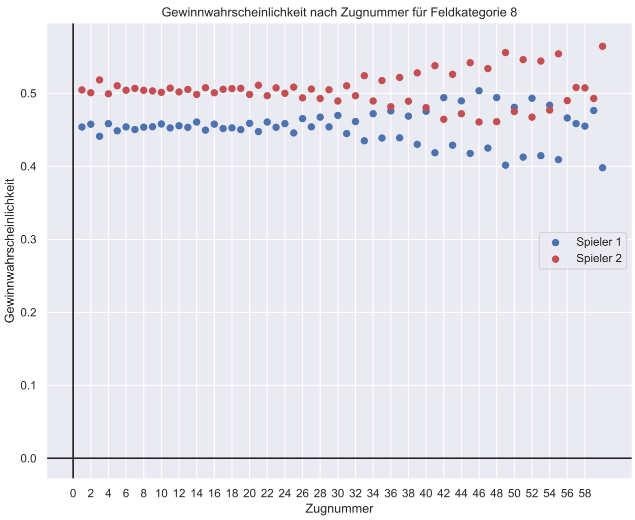


Abbildung 7.9: Gewinnwahrscheinlichkeit nach Zug für Feld-Kategorie 8

Abbildung 7.10: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 0

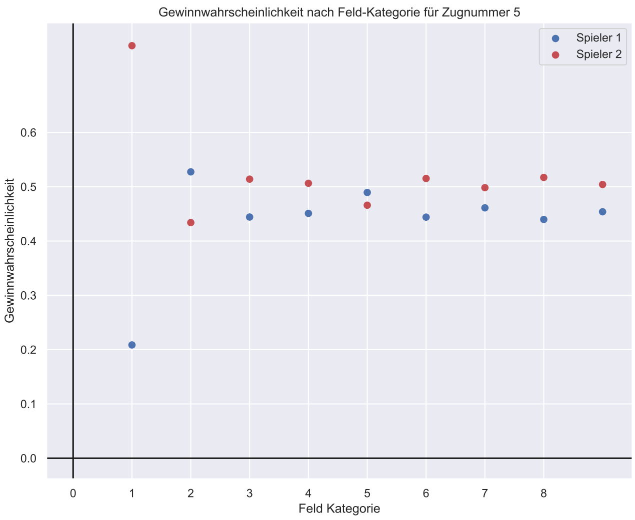


Abbildung 7.11: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 5

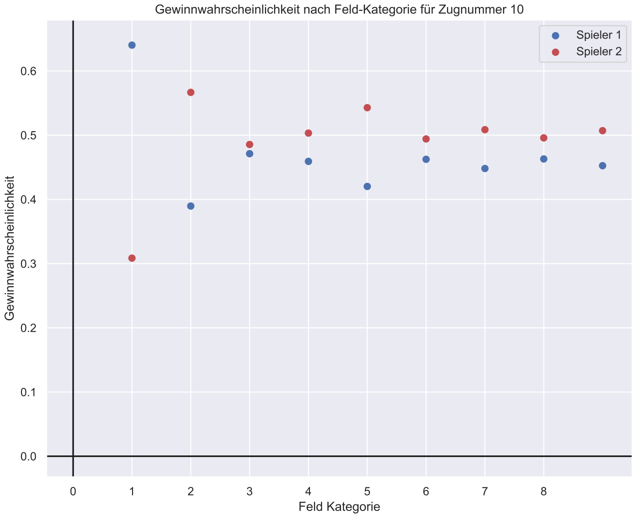


Abbildung 7.12: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 10

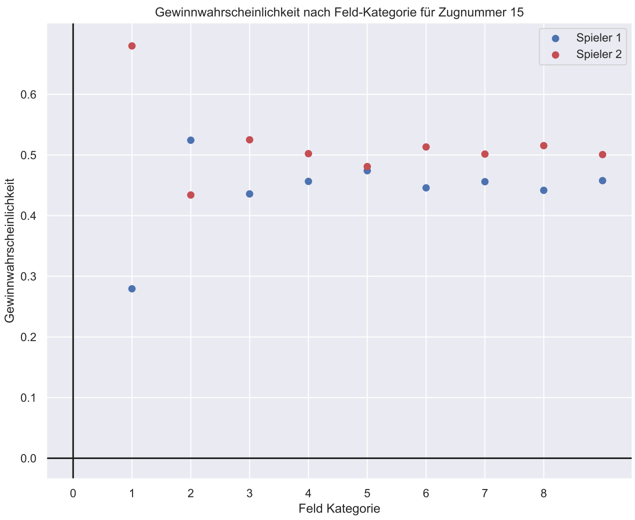


Abbildung 7.13: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 15

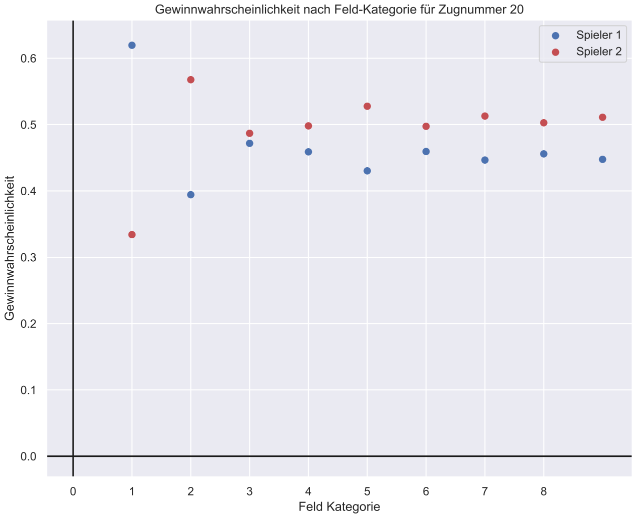


Abbildung 7.14: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 20

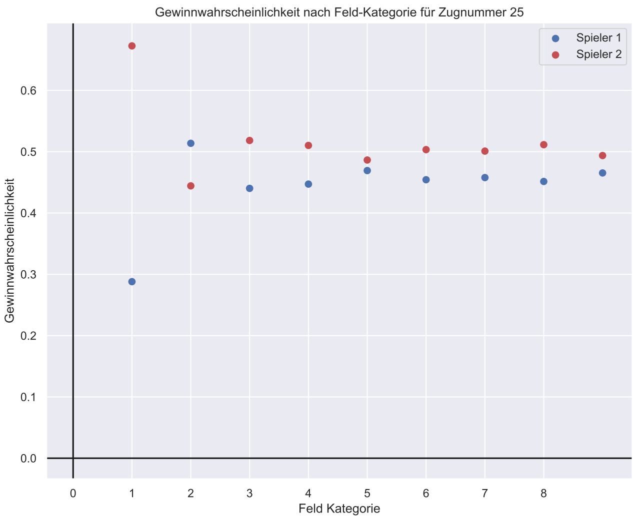


Abbildung 7.15: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 25

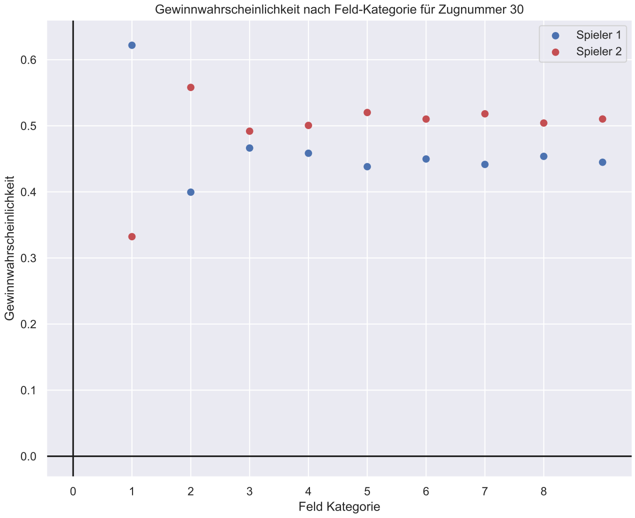


Abbildung 7.16: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 30

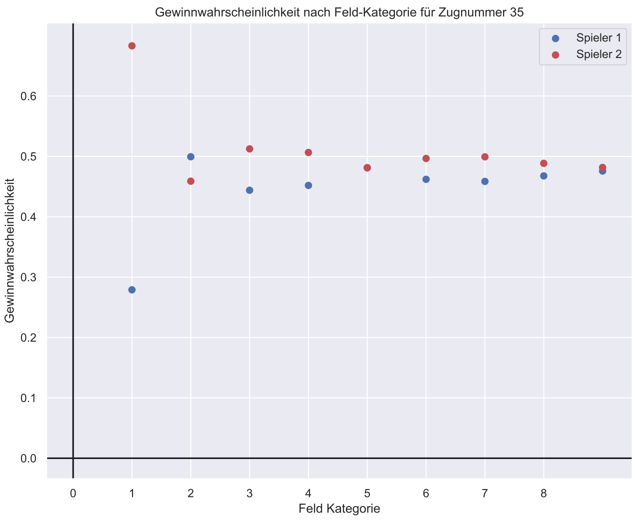


Abbildung 7.17: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 35

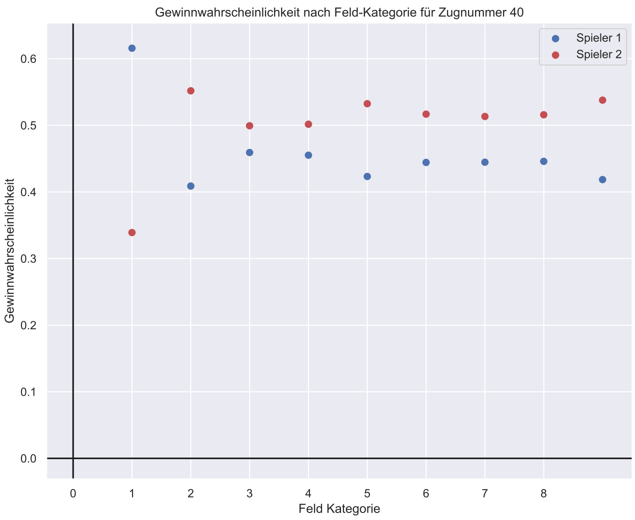


Abbildung 7.18: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 40

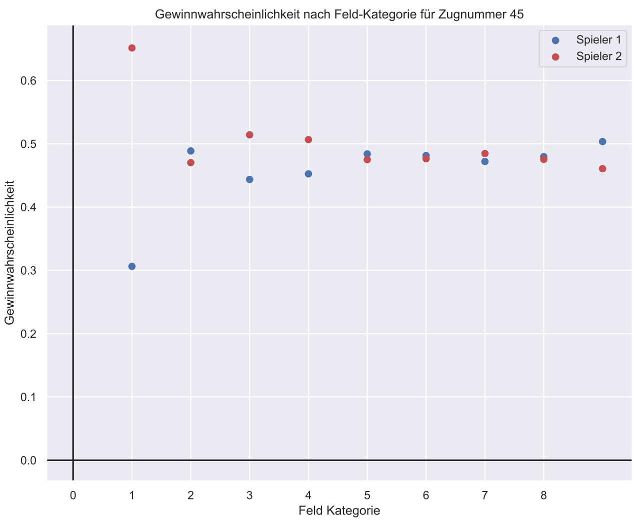


Abbildung 7.19: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 45

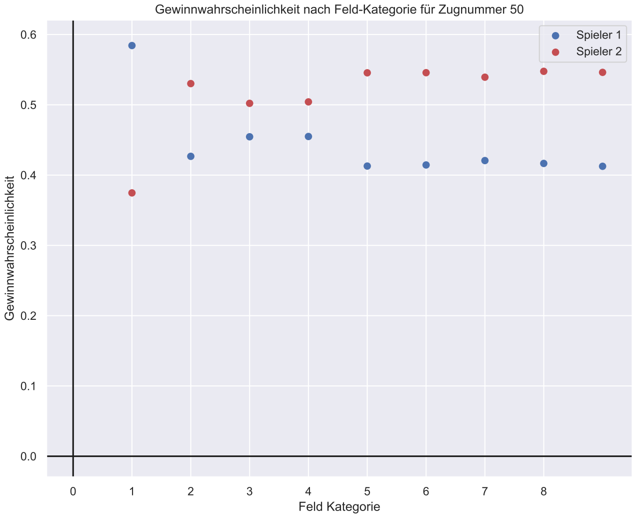


Abbildung 7.20: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 50

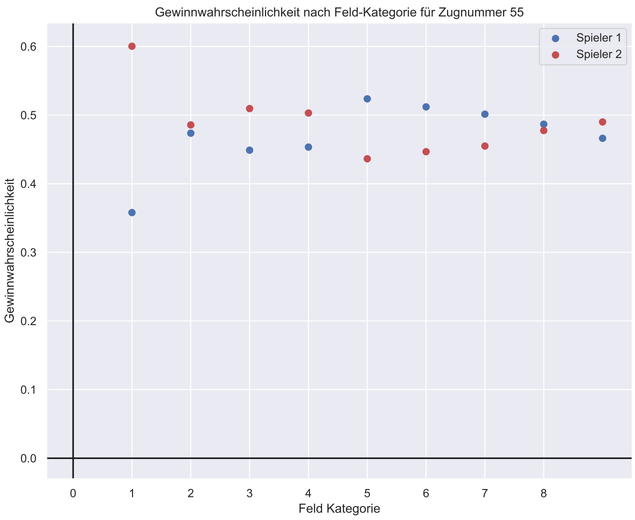


Abbildung 7.21: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 55

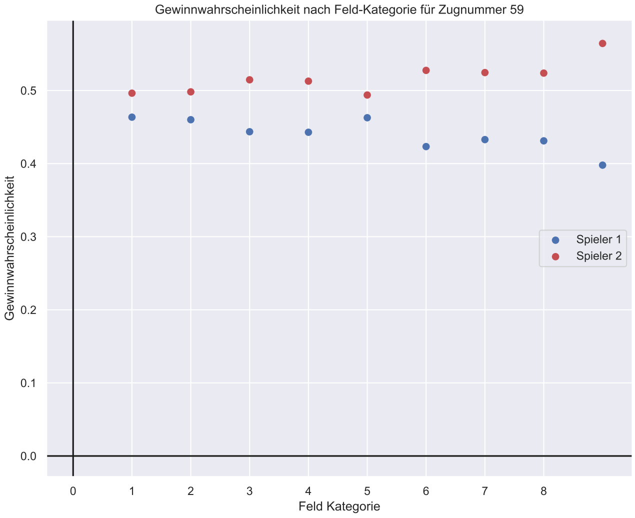


Abbildung 7.22: Gewinnwahrscheinlichkeit nach Feld Kategorie für Zug Nummer 59