Othello

— Entwicklung einer KI für das Spiel —

Patrick Müller, Max Zepnik

25. März 2019

Inhaltsverzeichnis

Kapitel 1

Einleitung

Computergegner ..

am Ende schreiben

. test..

text1 ...

auf Fazit beziehen?

Kapitel 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.V15]
- C-Felder: Felder, welche ein Feld horizontal oder vertikal von den Ecken entfernt sind (vgl. [Ber])
- X-Felder: Felder, welche ein Feld diagonal von den Ecken entfernt sind [o.V15]
- Zentrum: innerste Felder von C3 bis F6 (grüne Felder) [o.V15]
- Zentralfelder: Felder D4 bis E5 [o.V15]
- Frontsteine: die äußersten Steine auf dem Spielbrett um das Zentrum (vgl. [Ort]).

Diese Kategorien sind für die spätere Strategie wichtig.

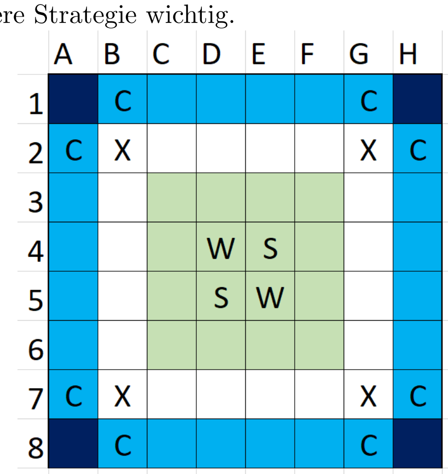


Abbildung 2.1: Kategorien des Spielfeldes

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 einfache Spielregeln, welche im Spielverlauf aber auch taktisches oder strategisches Geschick erfordern. Jeder Spieler legt abwechselnd einen Stein auf das Spielbrett. Dabei sind folgende Spielregeln zu beachten 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.
- Ist für beide Spieler kein Zug mehr möglich, ist das Spiel beendet. Der Spieler mit den meisten Steinen seiner Farbe gewinnt das Spiel.
- Das Spiel endet auch wenn alle Felder des Spielbrettes besetzt sind. In diesem Fall gewinnt ebenfalls der Spieler mit den meisten Steiner seiner Farbe.

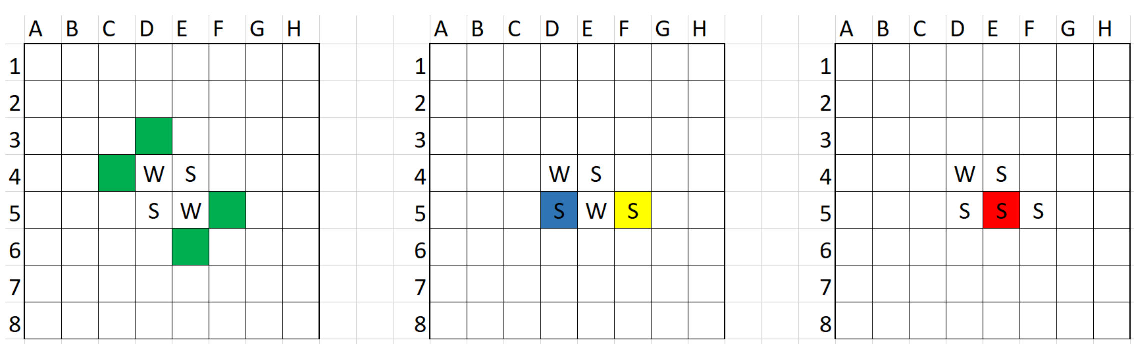


Abbildung 2.2: valide Zugmöglichkeiten für Schwarz und ausgeführter Zug

2.2 Spielverlauf

Das Spiel wird in drei Abschnitte eingeteilt [Ort]:

- Eröffnungsphase
- Mittelspiel
- Endspiel

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

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. [Ort]).

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 [Ort]

[Ort] 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.

Kapitel 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 [RN16] angelehnt.

Definition 1 (Spiel (Game)(vgl. [RN16] S. 162)) Ein Spiel besteht aus einem Tupel der Form

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

Definiere Q als 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 ($states$)
- $\mathtt{S}_0 \in \mathtt{Q}$ 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.
  $\mathtt{moves} = \{< X, Y > | X \in \{0..7\} \land Y \in \{0..7\}\}$
  Diese geben die Koordinate der Zugposition des neuen Steines auf dem Spielbrett an.
- $\mathtt{result} : \mathtt{Q} \times \mathtt{actions} \to \mathtt{Q}$ definiert das Resultat einer durchgeführten Aktion a und in einem Zustand s.

- $\texttt{terminalTest} : \texttt{Q} \to \mathbb{B}$ prüft ob ein Zustand s ein Terminalzustand ($s \in \texttt{terminalStates}$), also Endzustand, darstellt.
  $\texttt{terminalTest}(s) = \begin{cases} \texttt{True} & |s \in \texttt{terminalStates} \\ \texttt{False} & |s \notin \texttt{terminalStates} \end{cases}$
- $\texttt{utility} : \texttt{terminalStates} \times \texttt{Players} \to \{-1, 0, 1\}$ 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. [RN16] S. 161)) In einem Nullsummenspiel ist die Summe der utility Funktion eines Endzustandes ($\texttt{terminalState}$) 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. [RN16] 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. [RN16] 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 [RN16] S.164):

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

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

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

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 Search Trees voraus. Bei einem durchschnittlichen Verzweigungsfaktor von f bei einer Tiefe von d resultiert daraus eine Komplexität von $\mathcal{O}(d^{f})$. Bei einem einmaligen Erkunden der Knoten können die Werte aus den Blättern rekursiv von den Blättern zu den Knoten aktualisiert werden. Dadurch muss im nächsten Zug nur das Minimum aus `actions` ermittelt werden, da alle Kindknoten schon evaluiert wurden. Für übliche Spiele kann die 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.

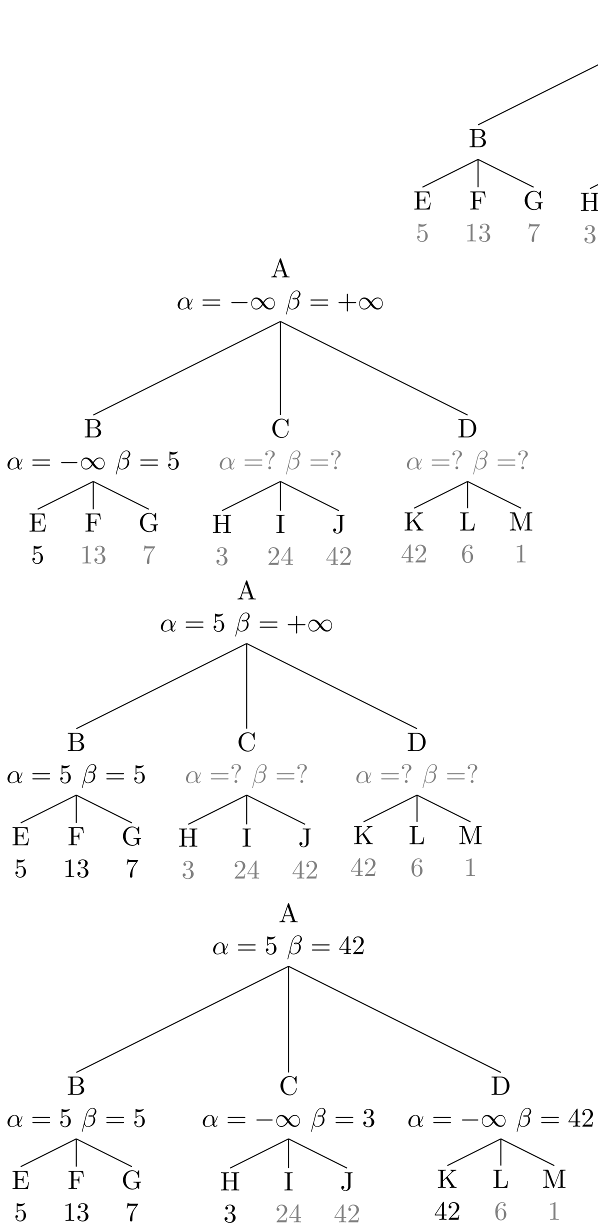
Bei der Betrachtung des Entscheidungsverhaltens des Algorithmus fällt jedoch schnell auf, dass ein nicht unerheblicher Teil aller möglichen Züge durch einen menschlichen Spieler gar nicht erst in Betracht gezogen wird. Dies geschieht aufgrund der Tatsache, dass diese Züge in einem schlechteren Ergebnis 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 [RN16] 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.

Abbildung 3.1: Beispielhafter Spielbaum



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 Aussageüber den schlechtesten Wert kann noch nicht getroffen werden.

Warum

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 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 [Str19]

Listing 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.

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 Minuten, 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. [RN16] 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. [RN16] 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. 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. [RN16] 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 [Nij07].

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.

Kapitel 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 Abhängigkeit vorhanden kann das Spiel durch 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“

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

```python
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.

Listing 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 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.

Listing 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:
26             self._last_turn_passed = True
27             self._prepare_next_turn()
28     else:
29         self._last_turn_passed = False
```

Weitere Funktionen der Klasse „Othello“ Die Klasse „Othello“ enthält weitere Funktionen auf die 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 Spieler-implementierungen 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 beiden Heuristiken implementiert:

1. Nijssen 2007 Heuristik

2. Stored Monte-Carlo-Heuristik

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

4.2.1 Nijssen 2007 Heuristik

Die „Nijssen 2007 Heuristik“ wurde aus [Nij07] 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 $b_f = \begin{cases} 1, \ \texttt{wenn F}(s)[f] = \texttt{player} \\ -1, \ \texttt{wenn F}(s)[f] = \texttt{other}(\texttt{player}) \\ 0, \ \texttt{sonst} \end{cases}$

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 Unterkapiteln **??** und **??** erklärt. Darauf aufbauend wird die Funktionsweise der Heuristik in dem Kapitel **??** erläutert.

Datenbank Die Datenbank speichert zu jeder Zugnummer die Gewinnwahrscheinlichkeiten der Spieler für Feldkategorien. 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.

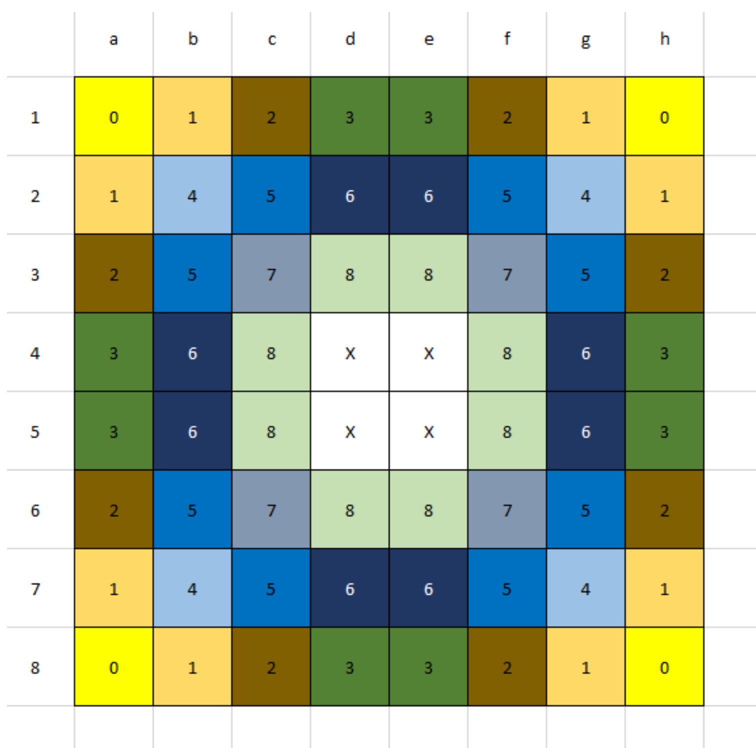


Abbildung 4.1: Symmetrie des Spielfeldes

Die Datenbank besteht aus 60 Zeilen und neun Spalten. Die neun Spalten stellen o.g. Feldkategorien („0“ bis „8“) dar. Die n-te Zeile stellt die Gewinnwahrscheinlichkeiten des Spielfeldes in dem n-ten Spielzug da.
Jede Zelle in dieser Tabelle enthält ein Tripel. 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 **??** erklärt.
Zwischen den drei Komponenten gibt es folgenden mathematischen Zusammenhang:
$\texttt{won\_games1} + \texttt{won\_games2} \leq \texttt{total\_played\_games}$
Durch unentschiedene Spiele kann die Gesamtanzahl der Spiele größer als die Summe der gewonnen Spiele der zwei Spieler sein. In Abbildung **??** 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 Kapitel initialisiert erstellt wurde, wird diese nun mit Spieldaten aus zufällig gespielten Spielen befüllt. In Listing **??** 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 **??** abgebildet.

Listing 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)
```

Listing 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

4.3 Start Tabellen

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 [?] 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.
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

Dieser Spieler verwendet den in Kapitel 3.3 verwendeten Algorithmus um den Zug mit der höchsten Gewinnwahrscheinlichkeit auszuwählen. Dazu spielt der Spieler zufällig eine bei Spielstart eingestellte Anzahl an Spielen ab der aktuellen Spielsituation und berechnet daraus den Anteil der gewonnen Spiele je verfügbaren Zug. Den Zug mit der höchsten Gewinnwahrscheinlichkeit wird nun im „realer“ Zug des Spielers ausgewählt.

Details zum Monte Carlo Spieler

In diesem Kapitel wird der Spieler Monte Carlo anhand des vorhandenen Quellcodes detailliert erklärt.
Die Spielerklasse „PlayerMonteCarlo“ ist in der Datei „python/Players/playerMonteCarlo.py“ zu finden. Die zunächst wichtigsten Funktionen sind die Init-Funktion der Klasse und die Funktion „get_move“.

init-Funktion In dem Konstruktor der Klasse „`PlayerMonteCarlo`“ werden mehrere Benutzerabfragen durchgeführt und in der Klasse gespeichert. Folgende Werte werden ermittelt:

- „`big_n`“: Anzahl der zufällig gespielten Spiele je Zug
- „`use_start_libs`“: Bool’scher Wert ob Startbibliotheken verwendet werden sollen.
- „`preprocessor`“: Nummer des verwendeten Preprozessors. 0 entspricht der Deaktivierung der Option
- „`preprocessor_parameter`“: Parameter des verwendeten Preprozessors
- „`heuristic`“: Wahl einer Heuristikfunktion

get_move „get_move“ ist eine Interface-Funktion, d.h. alle Spieler stellen eine Funktion „get_move“ bereit, die als Parameter einen Spielzustand (Klasse „`Othello`“) erwartet und einen „`move`“ zurückgibt. Dieser besteht aus einem Paar, das die Koordinaten auf dem Spielfeld darstellt. In Listing 4.4 ist die Funktion abgebildet.

In den Zeilen 3 bis 6 werden die Starttabellen verwendet, wenn diese in der init-Funktion ausgewählt wurden. Die Funktion „`get_available_start_tables`“ gibt eine Liste der möglichen Züge zurück (Z.4). Diese sind in der Form „a2“ angegeben. Aus den verfügbaren Zügen wird zufällig ein Element ausgewählt
(„`moves[random.randrange(len(moves))]`“ Z.6) und dieses in ein Koordinatenpaar übersetzt
(„`translate_move_to_pair`“ Z. 6).

Ist die Liste der verfügbaren Züge leer, oder die Nutzung der Starttabellen deaktiviert, wird der Quellcode ab der Zeile 8 ausgeführt. In den Zeilen 8 bis 18 werden Variablen implementiert und ggf. der Preprozessor (Z. 13f.) ausgeführt.
Die in der Init-Funktion angegeben „`big_n`“ Spiele werden in der Schleife in den Zeilen 20 bis 26 durchgeführt. „`simulated_game`“ ist eine Kopie des aktuellen Spielzustandes (Z. 21). Auf dieser Kopie wird nun ein zufälliges Spiel durchgeführt und gibt das Paar „`(first_played_move, won)`“ zurück (Z. 23). Das lokale Dictionary „`winning_statistics`“ (Z. 25f.) speichert diese Werte und summiert die Anzahl der gewonnen Spielen. Das Dictionary speichert für jeden Zug ein Paar, das die gewonnen Spiele und die Gesamtanzahl der Spiele darstellt. Nach dem Spielen aller zufälligen Spiele wird die Gewinnwahrscheinlichkeit berechnet (Z. 29 - 31). Die Wahrscheinlichkeit wird in dem Klassendictionary „`move_probability`“ gespeichert (Z. 31).
Abschließend wird das Maximum des Dictionary über die Wahrscheinlichkeiten ermittelt und zurückgegeben (Z. 33f.).

Listing 4.6: get_move Funktion des Alpha-Beta Spielers

```
1  def get_move(self, game_state: Othello):
2  
3      if self.use_start_lib and game_state.get_turn_nr() < 10: # check whether start move match
4          moves = self.start_tables.get_available_start_tables(game_state)
5          if len(moves) > 0:
6              return UtilMethods.translate_move_to_pair(moves[random.randrange(len(moves))])
7  
8      winning_statistics = dict()
9      self.move_probability.clear()
10 
11     own_symbol = game_state.get_current_player()
12 
13     if self.preprocessor is not None:
14         self.preprocessor(game_state, self.preprocessor_parameter, self.heuristic)
15 
16     possible_moves = game_state.get_available_moves()
17     for move in possible_moves:
18         winning_statistics[move] = (0, 1) # set games played to 1 to avoid division by zero error
19 
20     for i in range(self.big_n):
21         simulated_game = game_state.deepcopy()
22 
23         first_played_move, won = self.play_random_game(own_symbol, simulated_game)
24 
25         (won_games, times_played) = winning_statistics[first_played_move]
26         winning_statistics[first_played_move] = (won_games + won, times_played + 1)
27 
28 
29     for single_move in winning_statistics:
30         (games_won, times_played) = winning_statistics[single_move]
31         self.move_probability[single_move] = games_won / times_played
32 
33     selected_move = max(self.move_probability.items(), key=operator.itemgetter(1))[0]
34     return selected_move
```

4.4.4 Alpha-Beta Pruning

Ebenso wie der Spieler „Monte Carlo“ wird der Spielalgorithmus im Theorieteil erläutert. Der beste Zug wird dadurch berechnet, dass eine eingeschränkte Breitensuche bis zu einer bestimmten Tiefe durchgeführt wird, dabei allerdings auch der Gegenspieler beachtet wird. Statt einer kompletten Tiefensuche mit MiniMax werden Züge mit einer geringen Zugwahrscheinlichkeit nicht evaluiert. Die Grundidee des Algorithmus ist, dass sowohl der aktuelle Spieler, als auch der Gegenspieler jeweils den für sie besten Zug und für den Gegner schlechtesten Zug spielen.
Nach der eingeschränkten Breitensuche können mehrere Möglichkeiten gewählt werden. Es existieren einerseits mehrere Heuristiken, andererseits können auch andere Spieler ab diesen Spielzügen das Spiel berechnen. Diese Möglichkeiten werden in dem Kapitel **??** genauer erläutert.

Details zum Spieler Alpha-Beta Pruning

In diesem Kapitel wird der Spieler „Alpha-Beta Pruning“ anhand des vorhandenen Quellcodes detailliert erklärt. Die Spielerklasse „PlayerAlphaBetaPruning“ ist in der Datei „python/Players/playerAlphaBetaPruning.py“ zu finden. Die zunächst wichtigsten Funktionen sind die Init-Funktion der Klasse und die Funktion „get_move“.

Init-Funktion In dem Konstruktor der Klasse „`PlayerAlphaBetaPruning`“ werden folgende Benutzerabfragen getätigt:

- „`search_depth`“: Tiefe der Alpha-Beta Suche
- „`use_start_libs`“: Bool'scher Wert ob Startbibliotheken verwendet werden sollen.
- „`heuristic`“: Nummer der verwendeten Heuristik
- „`use_ml`“: Verwende statt der Heuristik Machine Learning in der Tiefe „`searchdepth`“ + 1
- „`use_monte_carlo`“: Verwende statt der Heuristik Monte Carlo in der Tiefe „`searchdepth`“ + 1
- „`ml_count`“: Anzahl der zufällig gespielten Spiele wenn Machine Learning oder Monte Carlo verwendet wird.

get_move In dem Listing 4.5 ist die „`get_move`“ Funktion des Alpha-Beta Pruning Spielers abgebildet. Zunächst wird ebenfalls geprüft, ob die Starttabellen verwendet werden sollen (Z. 3 - 6).
Ist keine passende Starttabelle verfügbar, wird ab Zeile 8 Alpha-Beta Abschneiden durchgeführt. Dies bedeutet,dass die Annahme getroffen wurde, dass jeder Spieler stets den besten Zug für sich, aber zeitgleich den schlechtmöglichsten Zug für den Gegner auswählt.
Der Algorithmus führt dadurch eine eingeschränkte Breitensuche bis zu einer bestimmten Tiefe durch. Anschließend wird ein Wert ermittelt, der angibt, wie „gut“ der gewählte Zug ist. Diese Funktion „`get_value`“ existiert in drei unterschiedlichen Varianten. Die erste Variante ist „`value()`“ (Z. 19), welche eine Heuristik zur Berechnung des Wertes verwendet. Die zweite Variante ist „`value_ml()`“ (Z. 14), welche ab diesem Spielzustand das Spiel mit dem Machine Learning Algorithmus spielt und die Gewinnwahrscheinlichkeit des besten Zuges zurückgibt. Die dritte Variante ist „`value_monte_carlo()`“ (Z. 16), welche ab diesem Spielzustand das Spiel mit dem Monte Carlo Algorithmus spielt und die Gewinnwahrscheinlichkeit des besten Zuges zurückgibt.

Listing 4.7: get_move Funktion des Alpha-Beta Spielers

```
def get_move(self, game_state: Othello):

    if self.use_start_lib and game_state.get_turn_nr() < 10: # check whether start move match
        moves = self.start_tables.get_available_start_tables(game_state)
        if len(moves) > 0:
            return UtilMethods.translate_move_to_pair(moves[random.randrange(len(moves))])

    best_moves = dict()
    for move in game_state.get_available_moves():
        next_state = game_state.deepcopy()
        next_state.play_position(move)

        if self.use_ml:
            result = -PlayerAlphaBetaPruning.value_ml(next_state,
                        self.search_depth - 1, ml_count=self.ml_count)
        elif self.use_monte_carlo:
            result = -PlayerAlphaBetaPruning.value_monte_carlo(next_state,
                        self.search_depth - 1, self.heuristic, mc_count=self.ml_count)
        else:
            result = -PlayerAlphaBetaPruning.value(next_state, self.search_depth - 1, self.heuristic)

        if result not in best_moves.keys():
            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]))]
```

Kombination des Alpha-Beta Pruning Agenten mit weiteren Agenten

Kapitel 5

Evaluierung

Kapitel 6

Fazit

Notes

Literaturverzeichnis

[Ber] Berg, Matthias. Strategieführer. http://berg.earthlingz.de/ocd/strategy2.php. [Online; accessed 20-January-2019].

[Nij07] J. A. M. Nijssen. Playing othello using monte carlo, Jun 2007.

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

[o.V15] o.V. Spiele: Othello. https://de.wikibooks.org/wiki/Spiele:_Othello, 2015. [Online; accessed 20-January-2019].

[RN16] Stuart J. Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Always learning. Pearson, 2016.

[Str19] Prof. Dr. Karl Stroetmann. An introduction to arti

cial intelligence - lecture notes in progress. 2019.