Systematischer Vergleich von Alpha-Beta und Monte-Carlo Tree Search für das Spiel Othello

Bachelorarbeit

Alessio Mossudu
390642

24. Oktober 2023

Betreuer: Prof. Dr. Benjamin Blankertz
Prof. Dr. Marc Alexa

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Kurzfassung

Othello ist ein vergleichsweise neues aber etabliertes Brettspiel für zwei Spieler mit strategischen Elementen. Die Komplexität stellt eine Herausforderung bei der Entwicklung eines Programms zur Ermittlung des besten Zugs dar. Das Ziel dieser Arbeit ist es potenzielle Lösungsansätze im Kontext von Othello, für diese Herausforderung zu präsentieren, zu implementieren und zu vergleichen. Es wurden hierbei die 3 Algorithmen „Minimax“ mit verschiedenen Optimierungen und Bewertungsfunktionen, „Monte Carlo Tree Search“ mit unterschiedlichen Verbesserungen und ein kombinierter Algorithmus, der die Bewertungsfunktion von Alpha-Beta und die asymmetrische Baumsuche von MCTS kombiniert, welcher in dieser Arbeit auch „MCTS EVAL“ genannt wird, gegenübergestellt und verglichen. Hierfür wurden in einer Reihe von Experimenten Daten zu wichtigen Kriterien wie Suchtiefe, Anzahl durchsuchter Knoten und Spielqualität gesammelt und ausgewertet. Die Ergebnisse zeigen, dass „Monte Carlo Tree Search“ für Othello, wegen der Komplexität der Züge, nicht geeignet ist. Dafür dominiert „MCTS EVAL“ Alpha-Beta in seiner besten Form, vor Allem bei geringer Rechenzeit pro Zug.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

MCTS Monte Carlo Tree Search

MCTS EVAL Algorithmus: MCTS mit Bewertungsfunktion an Stelle der Monte Carlo Simulation

MCTS EVAL RING Algorithmus: MCTS EVAL mit Ring-Optimierung

MCTS PURE Algorithmus: MCTS

MCTS DIRECTED Algorithmus: MCTS mit erweiterter UCT Funktion

MCTS RING Algorithmus: MCTS mit Ring-Optimierung

MCTS PARALLEL Algorithmus: MCTS mit 10 parallelen Simulationen pro Knoten

MCTS SEQUENTIAL Algorithmus: MCTS mit 10 sequentiellen Simulationen pro Knoten

MCTS MULTI COM Algorithmus: Kombination aus MCTS PARALLEL und MCTS SEQUENTIAL 6.3

Pure AB Algorithmus: Alpha-Beta ohne weitere Optimierungen

AB ITD Algorithmus: Alpha-Beta mit iterativer Tiefensuche

AB ITD TT Algorithmus: Alpha-Beta mit iterativer Tiefensuche und Transposition Tables

AB ITD R Algorithmus: Alpha-Beta mit iterativer Tiefensuche und Ring-Optimierung

1 Einleitung

Brettspiele wie Othello oder Schach sind seit Jahrzehnten Thema in der Erforschung von Spielbaum Suchalgorithmen. Diese sind wegen ihrer Komplexität und der daraus resultierenden Größe des, mit jeder weiteren Tiefe exponentiell wachsenden Suchbaums, einfach nicht möglich lösbar[13]. Dieses Problem hat dazu geführt, dass Algorithmen mit verschiedenen Ansätzen entwickelt wurden, welche mithilfe von Bewertungsfunktionen und Wahrscheinlichkeitstheorie, die bestmöglichen Züge in jeder Position finden sollen.

Während klassische Suchalgorithmen wie MiniMax mit Alpha-Beta Pruning seit Jahrzehnten dieses Forschungsfeld dominieren, erleben modernere probabilistische Suchalgorithmen wie Monte Carlo Tree Search in den letzten Jahren einen Aufschwung. Jüngstes Beispiel dafür ist die von Google entwickelte KI AlphaGo, welche 2015 die erste Go-KI war, die einen Meister 5-0 schlug.[1]

Diese Bachelorarbeit befasst sich mit dem Vergleich von Minimax, Monte Carlo Tree Search und einem kombinierten Algorithmus, der im Rahmen dieser Arbeit MCTS mit Bewertungsfunktion genannt wird, in Bezug auf deren Effizienz und Performanz auf dem Spiel Othello. Ziel ist es zu untersuchen welche Optimierungen, wie zum Beispiel iterative Tiefensuche oder Transposition Tables, sich wie auf die Algorithmen auswirken und am Ende zu sehen, welcher der Algorithmen sich als überlegen erweist.

Othello stellt zwei besonders interessante Problematiken dar. Eine der Strategien um einen guten Spielzustand zu erhalten ist es die eigene Mobilität zu maximieren und die des Gegners einzuschränken[12]. Das erhöht den eigenen Suchaufwand, was zu einer geringeren Suchtiefe führen kann. Hinzu kommt, dass anders als bei klassischen Brettspielen wie Schach, ein Zug in Othello nicht nur aus dem Bewegen oder Setzen einer einzelnen Spielfigur besteht, sondern immer mehrere Steine auf dem Feld von einem Zug betroffen sind. Das erhöht den Rechenaufwand für die Algorithmen.

Anhand mehrerer Tests und Experimente auf zufällig generierten Spielzuständen wird in dieser Arbeit festgestellt welcher der vorgestellten Algorithmen Othello am besten spielen kann, welche Faktoren die Bewertungsfunktion am meisten beeinflussen und welche algorithmischen Optimierungen sich tatsächlich lohnen.

2 Othello

Das Spiel Othello wurde 1971 von GORŌ HASEGAWA in Japan erfunden und ist eine Variante des Brettspiels Reversi, welches von LEWIS WATERMAN in den 1880ern in England entwickelt wurde [4][10]. Der Unterschied zu Reversi liegt darin, dass man bei Othello mit einer vorgeschriebenen Aufstellung der Steine beginnt. [6] In Europa blieb der Erfolg aus. In Japan und in den U.S.A. hingegen ist Othello weitaus bekannter und gehört zu einem der beliebtesten Brettspiele Japans.

Kompetitives Othello

Das erste offizielle Othello-Turnier fand 1973 in Japan statt und die erste Weltmeisterschaft (WOC) dann 4 Jahre später in Tokyo. Anfangs traten nur die Nationalmeister gegeneinander an, aber seit 1987 darf jedes Land bis heute noch mit 3 Spielern an der Weltmeisterschaft teilnehmen.[7]
1997 schlug das von Michael Buro entwickelte Programm Logistello den amtierenden Weltmeister TAKESHI MURAKAMI in 6 von 6 Spielen.

2.1 Regeln

Othello ist ein Zwei-Personen-Nullsummenspiel mit perfekter Information, welches auf einem 8×8-Brett gespielt wird. Die Spielsteine sind runde Plättchen, die auf einer Seite weiß und auf der anderen schwarz gefärbt sind.
Zu Beginn jedes Spiels werden vier Spielsteine in die in Abbildung 1 gezeigte vorgeschriebene Aufstellung gelegt, wobei schwarz beginnt.[5]

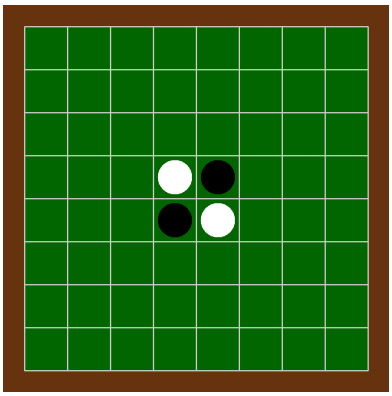

Abbildung 2.1: Startaufstellung

Die Spieler legen abwechselnd Steine mit ihrer eigenen Farbe nach oben auf ein leeres Feld, das horizontal, vertikal oder diagonal an ein bereits vom Gegner belegtes Feld angrenzt. Dabei muss sich

mindestens ein gegnerischer Stein in einer geraden Linie zwischen dem gerade gelegten und einem anderen Stein der eigenen Farbe befinden. Das nennt man „Einkreisen“ der Steine. Es können mit demselben Zug Steine in mehreren Richtungen, also zu Beispiel einer links und einer rechts von dem gerade gelegten Spielstein eingekreist werden. Die eingekreisten Steine werden anschließend umgedreht, sodass sie mit der eigenen Farbe nach oben liegen.[5]

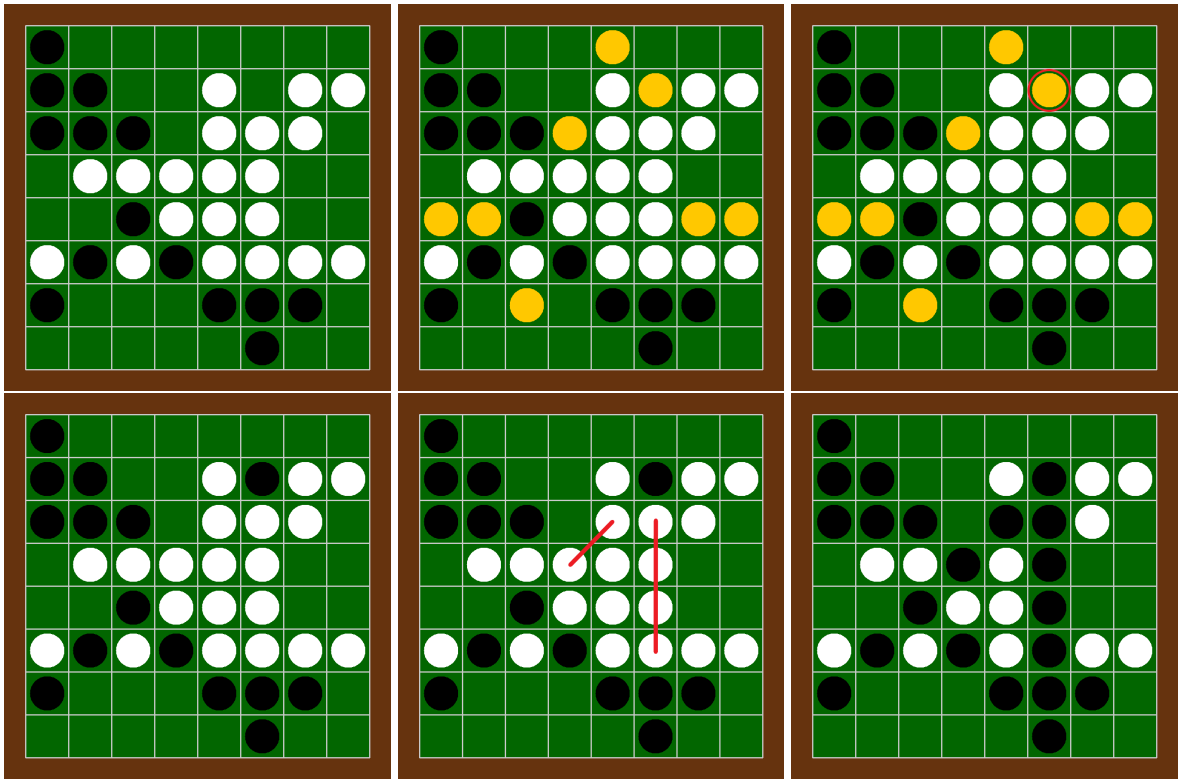

Abbildung 2.2: Ablauf eines Zuges in Othello. Die gelb markierten Felder stellen die möglichen Züge für schwarz dar. Schwarz wählt setzt seinen Stein auf das rot eingekreiste Feld. Die weißen durchgestrichenen Steine sind die umzudrehenden Steine.

Spielzüge sind nur zulässig, wenn sie zum Umdrehen von gegnerischen Steinen führen. Wenn ein Spieler also keinen zulässigen Zug hat, dann muss er passen.[5]

Es wird so lange gespielt, bis entweder das Brett voll ist oder beide Spieler keine Züge mehr machen können. Der Spieler, der am Ende die meisten eigenen Steine auf dem Brett hat gewinnt. Spiele können auch unentschieden enden.[5]

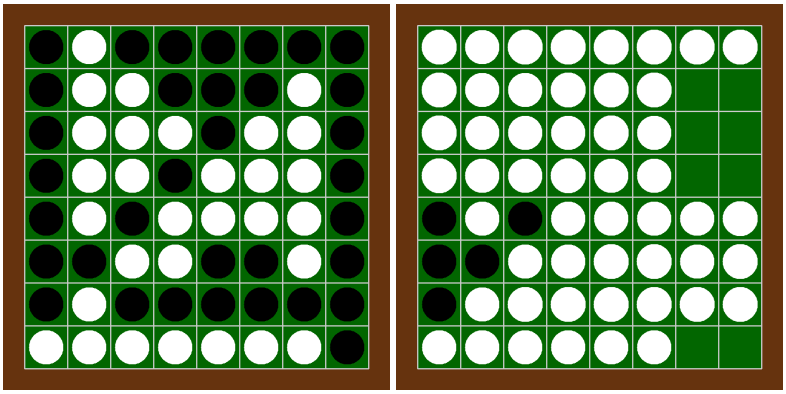

Abbildung 2.3: Mögliche Endstellungen

2.2 Spielbäume

Othello ist wie so viele kompetitive Brettspiele ein Zwei-Personen-Nullsummenspiel mit perfekter Information. Bei Zwei-Personen-Nullsummenspielen entsprechen die Verluste des verlierenden Spielers 1 zu 1 den Gewinnen des siegenden Spielers. Das heißt, dass ein Zug genauso gut für Spieler A gut ist, wie er für Spieler B schlecht ist. Perfekte Information heißt, dass beide Spieler zu jedem Zeitpunkt alle Informationen zum Spielzustand kennen.[17]

Die Zugmöglichkeiten eines solchen Spieles lassen sich ganz gut als Spielbaum darstellen. Ein Spielbaum ist in der Spieltheorie ein azyklischer gerichteter Graph. Dessen Knoten stellen dabei die einzelnen Spielzustände und dessen Kanten die Spielzüge dar. Obwohl der gleiche Spielzustand mehrmals durch eine andere Zugabfolge erreicht werden kann, nutzt man der Einfachheit halber dennoch einen azyklischen Graph. Um eine redundante Baumsuche zu vermeiden können sogenannte Transposition Tables genutzt werden[2]. Darauf wird in Kapitel 3.2 näher eingegangen.

In jedem Knoten werden mindestens der Zustand des Spielfelds und der Spieler, welcher am Zug ist gespeichert. Wie es sich für einen Suchbaum gehört, kann jeder Knoten Nachfolger haben oder ein Endknoten sein. Knoten ohne Nachfolger werden auch Blattknoten genannt.

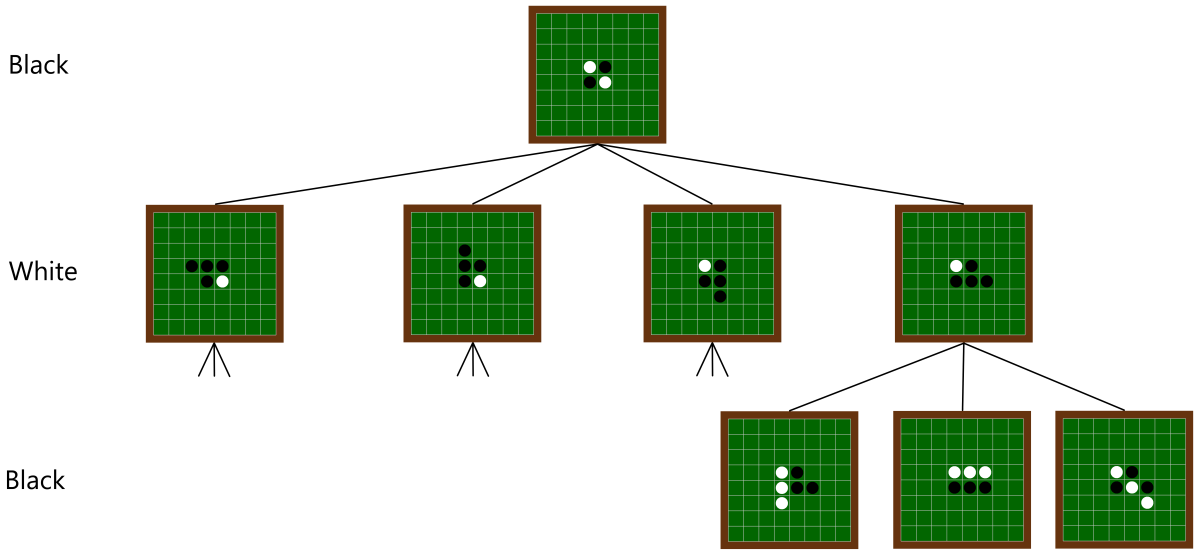

Abbildung 2.4: Othello Spielbaum

Othello hat zusätzlich die Eigenschaft, dass das Spiel einen endlichen Suchbaum hat und spätestens nach 60 Zügen, wenn alle Steine auf dem Spielfeld liegen, endet. Das macht es möglich Algorithmen wie Monte Carlo Tree Search für die Suche nach dem besten Zug zu nutzen.

3 Minimax

Die Minimax Suche ist ein rekursiver Tiefensuche-Algorithmus, der den gesamten Spielbaum expandiert und durchsucht. Die Blattknoten werden evaluiert und die Werte werden nach dem Minimax Prinzip über die Elternknoten bis an die Wurzel weitergegeben. Den gesamten Suchbaum aufzuspannen ist meistens nicht realisierbar, da die Anzahl der Knoten mit jeder weiteren Tiefe exponentiell wächst. [11]

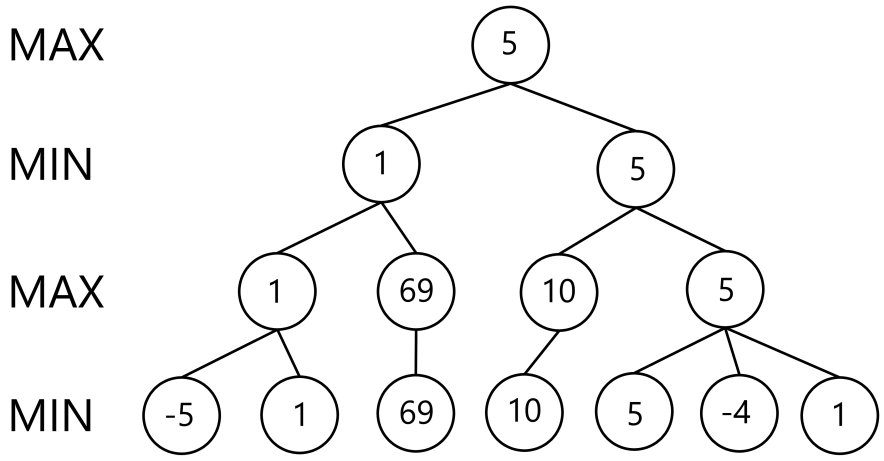

Abbildung 3.1: Minimax

Deshalb limitiert man die Suche meist bis zu einer gegebenen Tiefe. Auf die Zustände in den Blattknoten wird dann eine sogenannte Bewertungsfunktion angewandt um eine numerische Abschätzung der Position auszurechnen und weiterzugeben.[11]

3.1 Alpha-Beta-Pruning

Das Alpha-Beta-Pruning ist eine Optimierung der Minimax Suche, die darauf abzielt die Anzahl der zu durchsuchenden Knoten zu reduzieren. Eine sogenannte Beschneidung findet dann statt, wenn dem Algorithmus klar ist, dass das Durchsuchen eines Teilbaums zu keiner neuen Erkenntnis führen würde. Es liefert genauso wie der Minimax-Algorithmus den optimalen Zug. [11]

Im linken Teilbaum findet ein Beta-Cutoff statt. Der zuerst durchsuchte MAX-Knoten auf Tiefe 2 erhält eine Evaluation von 10. Dem zweiten MAX-Knoten auf dieser Ebene wird ein Beta Wert von genau dieser Evaluation mitgegeben. Diesem wird aus Tiefe 3 den Wert 12 übergeben. Da dieser größer ist als der Beta Wert und der Elternknoten ein MIN-Knoten ist, muss der zweite MAX-Knoten

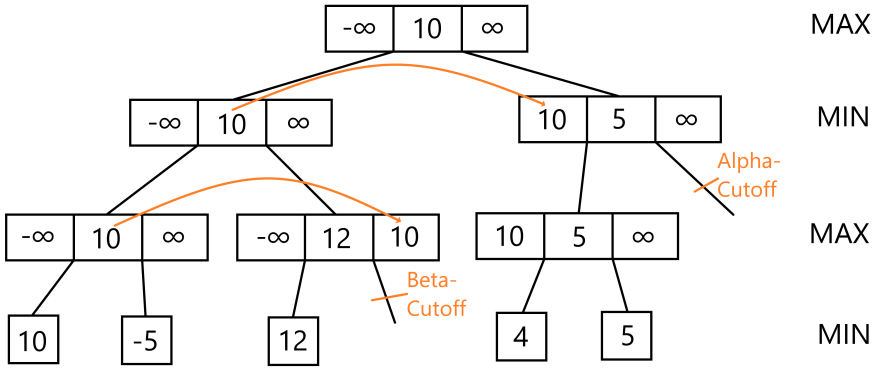

Abbildung 3.2: Minimax mit Alpha-Beta Pruning

nicht weiter durchsucht werden. Im rechten Teilbaum läuft das ganze umgekehrt analog ab. Hier wird dem zweiten MIN-Knoten in Tiefe 1 der Alpha Wert 10 übergeben. Da bereits das Durchsuchen des ersten Kindknotens den Wert 5 liefert, der kleiner als der Alpha-Wert ist, muss auch dieser Knoten nicht weiter betrachtet werden.

Im Optimalfall sind die Züge bereits so vorsortiert, dass zuerst immer der beste und zuletzt der schlechteste Zug durchsucht werden muss. Das würde zu enormen Beschneidungen des Suchbaums führen. Im schlimmsten Fall entspricht Laufzeit von Alpha Beta der Laufzeit vom Minimax Algorithmus, und zwar $O(b^d)$, wobei b der durchschnittliche Verzweigungsfaktor und d die Tiefe ist.[11] Bei perfekter Vorsortierung der Züge entspricht die Laufzeit $O(b^{d/2})$.[11]

3.2 Alpha-Beta Optimierungen

Man kann mithilfe weiterer Optimierungen den Suchbaum noch weiter beschneiden, was es dem Algorithmus ermöglicht größere Tiefen zu erreichen. Im folgenden Abschnitt werden 2 solcher bekannten Optimierungen und eine selbst erstellte Optimierung vorgestellt.

Iterative Tiefensuche

Man kann vor dem Start des Algorithmus‘ nicht ausrechnen, welche die maximale Suchtiefe ist, die in einem gegebenen Zeitlimit erreicht werden kann. Bei der iterativen Tiefensuche wird die Suchtiefe Schritt für Schritt erhöht und für jede Suchtiefe ein Ergebnis ausgerechnet. Die für die einzelnen Knoten berechneten Evaluationen und ihre Spielpositionen werden festgehalten und können in späteren Iterationen zur Vorsortierung der Züge genutzt werden, was wiederum zu mehr Alpha und Beta Cutoffs führt. Othello hat einen Verzweigungsfaktor b von etwa 10.[14] Alle Bäume, die vor einer Tiefe t durchsucht werden, enthalten zusammen $\frac{1}{(b-1)} = 1/9$ der Knoten, die in Tiefe t enthalten sind.[11]

Transposition Tables

Es kann durchaus vorkommen, dass der gleiche Spielzustand an mehreren Stellen im Suchbaum erreicht wird. Das kann man sich zu Nutze machen, indem man jede ausgerechnete Position und ihre Evaluation sowie die Tiefe in der sie übergeben wurde, in einer sogenannten Transposition Table festhält. Wenn der Algorithmus beim Durchlaufen des Suchbaums wieder auf die gleiche Position trifft und die Tiefe des aktuellen Zustands größer oder gleich der Tiefe des Zustands aus der Transposition Table ist, dann muss der Algorithmus nicht weiter den Baum an dieser Stelle durchsuchen, sondern kann einfach die Evaluation aus der Transposition Table zurückgeben.[16] Das lohnt sich besonders in Spielen mit relativ einfachen Zügen, bei denen zum Beispiel nur eine Figur bewegt wird und in Spielen bei denen Figuren auch wieder zurück bewegt werden können. Schach ist ein perfektes Beispiel dafür: das Spiel hat zwar einen sehr großen Verzweigungsfaktor, viele Positionen können aber durch die Veränderung der Zugreihenfolge mehrmals erreicht werden.[2]

Zugring

Der Zugring ist eine eigene Kreation, um die Suche nach legalen Spielzügen zu beschleunigen. Vor dem Start des Alpha-Beta Algorithmus‘ wird ein Ring um die bereits gelegten Steine erstellt und in einem 2-Dimensionalen BOOLEAN-Array gespeichert. Beim Durchlaufen des Suchbaums wird dieser bei jedem Schritt vergrößert oder verkleinert. Mit Hilfe des Rings muss nicht das ganze Spielfeld nach möglichen Zügen durchsucht werden, sondern nur noch dieser sich and das Spielfeld anpassende Ring an Feldern.

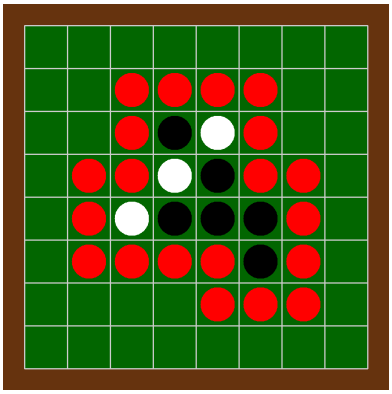

Abbildung 3.3: Zugring: rot markierte Felder

3.3 Evaluation

Die Bewertungsfunktion erlaubt es dem Algorithmus jedem Spielzustand einen numerischen Wert sowohl aus Sicht des Spielers, als auch aus Sicht des Gegners zu geben. Dies ist ein heuristischer Wert, der aus einem oder auch mehreren Faktoren zusammengesetzt werden kann. Ein positiver Wert bedeutet, dass die Position für den Spieler, aus dessen Perspektive das Spielfeld betrachtet wird, gut ist. Eine Null bedeutet, dass die Position für beide Spieler als gleich gut gesehen wird.

Das Ziel bei Othello ist es am Ende die meisten Steine der eigenen Farbe auf dem Brett zu haben, jedoch ist es meistens vom Nachteil schon während des Spiels viele Steine der eigenen Farbe zu haben.[3] Es gibt einige Strategien, die einem helfen können, ein Spiel zu gewinnen. Für den Aufbau der einzelnen Faktoren der Bewertungsfunktion orientiert sich diese Arbeit an dem Paper „An Analysis of Heuristics in Othello“[15] von VAISHNAVI SANNIDHANAM und MUTHUKARUPPAN ANNAMALAI.

Ecken

Steine, die in die Ecken des Spielbretts gesetzt werden, können nicht mehr umgedreht werden, sie sind also stabil.[15] Indem man einen Stein in eine Ecke platziert, macht man es dem Gegner schwerer Steine auf dem Rand des Spielbretts und auf der zur Ecke gehörenden Diagonale zu platzieren, da man immer die Möglichkeit hat diese wieder umzudrehen.
Es macht also Sinn im Verlaufe des Spiels die Ecken einzunehmen. Der Besitz mehrerer Ecken führt aber nicht zwangsweise zum Sieg, der Gegner kann nämlich die Stärke einer Ecke schwächen, indem dieser einfach um die Ecken herum spielt. Die Bewertung errechnet sich einfach aus der Summe der Ecken im Besitz des Spielers und des Gegners.

Algorithm 1 cornersEval

```
maxCorners = getNumberOfCornersCaptured(turnplayer)
minCorners = getNumberOfCornersCaptured(oppositePlayer)

if maxCorners + minCorners ≠ 0 then
    return 100*(maxCorners - minCorners)/(maxCorners + minCorners)
else
    return 0
end if
```

Mobilität

Einer der wichtigsten Strategien, die es zu verfolgen gilt ist die Mobilitätsstrategie. Man möchte immer die Anzahl der eigenen Züge maximieren und die des Gegners minimieren. Eine Auswahl aus einer kleinen Menge an Zügen kann zum sogenannten Zugzwang führen (ein Konzept, das man vom Schach kennt), wobei ein Spieler gezwungen ist einen schlechten Zug zu auszuführen.[8]
Dafür werden auf dem aktuellen Spielbrett die Anzahl der legalen Züge sowohl für den Spieler, als auch für den Gegner ausgerechnet und gegenübergestellt.

Algorithm 2 mobilityEval

```
1: maxMobility = getNumberOfPossibleMoves(turnplayer)
2: minMobility = getNumberOfPossibleMoves(oppositePlayer)
3:
4: if maxMobility + minMobility ≠ 0 then
5:     return 100*(maxMobility - minMobility)/maxMobility + minMobility)
6: else
7:     return 0
8: end if
```

Flankenstrategie

Bei der potentiellen Mobilität, auch Flankenstrategie genannt, geht es darum die Anzahl der eigenen flankierbaren Steine zu minimieren um die zukünftige Mobilität des Gegners einzuschränken.
Die wird berechnet indem man für beide Spieler die Anzahl der freien Felder, die an einen eigenen Stein angrenzen, aufsummiert und wie bei der Mobilität mit einander verrechnet.

Algorithm 3 flanksEval

```
1: maxFlanks = 0
2: minFlanks = 0
3:
4: while not all discs checked do
5:     if disc to check belongs to turnplayer then
6:         minFlanks += getFlanksOfDisc()
7:     else
8:         maxFlanks += getFlanksOfDisc()
9:     end if
10: end while
11: if maxFlanks + minFlanks ≠ 0 then
12:     return 100*(maxFlanks - minFlanks)/maxFlanks + minFlanks)
13: else
14:     return 0
15: end if
```

Statische Bewertung

Für die statische Bewertung nimmt man das Spielbrett und weist jedem Feld einen Wert zu. Das kann als Matrix dargestellt werden. Hierfür wurde die Zuweisung des Papers „An Analysis of Heuristics in Othello“[15] übernommen.

Diese Matrix erfasst die wichtigsten Aspekte des Spiels wie zum Beispiel die Wichtigkeit der Ecken. Die statische Bewertung errechnet sich indem die Steine und ihre in der Matrix stehenden Gewichte ganz einfach aufsummiert werden

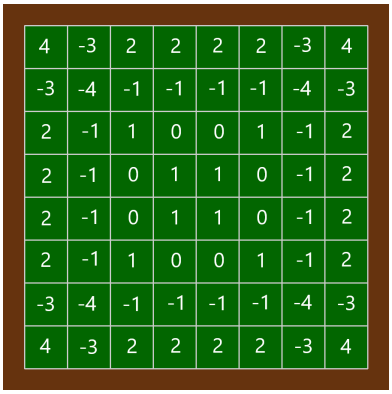

Abbildung 3.4: Statische Bewertungsmatrix

Algorithm 4 staticEval

```
1: eval = 0
2: while not all discs checked do
3:     if disc to check belongs to turnplayer then
4:         eval += getDiscEvalFromMatrix()
5:     else
6:         eval -= getDiscEvalFromMatrix()
7:     end if
8: end while
9: return eval
```

Stabilität

Stabile Steine nennt man Steine, die nicht mehr vom Gegner eingenommen werden können, wie zum Beispiel Steine, die man in eine der vier Ecken platziert. Steine mitten auf dem Spielfeld können aber auch stabil sein. Spielsteine, die der Gegner direkt im nächsten Zug einnehmen könnte nennt man instabil und alle anderen werden semi-stabil genannt.

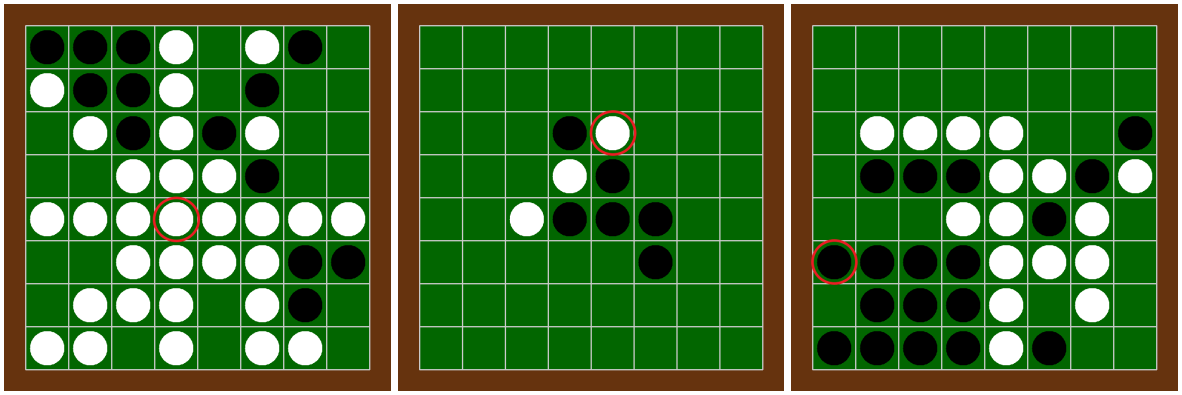

Abbildung 3.5: Stabilität einzelner Steine: links stabil, mitte instabil, rechts semi stabil

Für die Berechnung dieses Faktors wird zu erst jeder Kategorie ein Wert zugeordnet:

- Stabiler Stein = +1
- Instabiler Stein = -1
- Semi-stabiler Stein = 0

Die Stabilität aller Steine auf dem Spielfeld wird bestimmt und dessen Werte für die jeweiligen Spieler aufsummiert. Die sich daraus ergebenden Gesamtstabilitäten werden wie auch schon bei der Mobilität und Flankenstrategie verrechnet.

Algorithm 5 stabilityEval

```
 1: maxStability = 0
 2: minStability = 0
 3:
 4: while not all discs checked do
 5:     if disc to check belongs to turnplayer then
 6:         minStability += getStabilityOfDisc()
 7:     else
 8:         maxStability += getStabilityOfDisc()
 9:     end if
10: end while
11:
12: if maxStability + minStability ≠ 0 then
13:     return 100*(maxStability - minStability)/maxStability + minStability)
14: else
15:     return 0
16: end if
```

X- und C-Felder

Die Felder um die Ecken herum werden X- und C-Felder genannt und sind für gewöhnlich zu vermeiden, da sie dem Gegner die Möglichkeit geben einen Stein in eine Ecke zu platzieren. Diese werden auch in der statischen Bewertung sehr negativ gewichtet.

Das Einnehmen eines Feldes um eine Ecke herum wird mit einem negativen Wert bestraft, sodass der Vorteil, der dadurch erreicht wird, wirklich groß sein muss, um das Platzieren eines Steins in eines dieser Felder rechtzufertigen. Für ein eingenommenes C-Feld werden 5 von der gesamten Evaluation abgezogen und für ein eingenommenes X-Feld sogar 10.

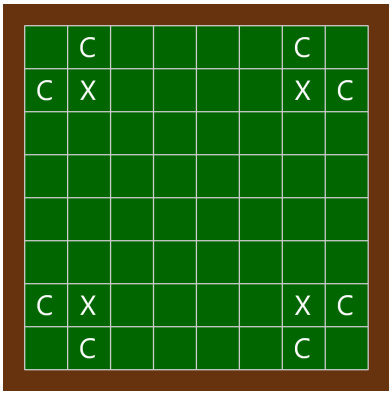

Abbildung 3.6: X- und C-Felder

Algorithm 6 xcfieldsEval

```
1: eval = 0
2: eval -= 10*numberOfXfieldsCaptured(turnplayer)
3: eval += 10*numberOfXfieldsCaptured(oppositePlayer)
4: eval -= 5*numberOfCfieldsCaptured(turnplayer)
5: eval += 5*numberOfCfieldsCaptured(oppositePlayer)
6:
7: return eval
```

Anzahl der Steine

Obwohl es anfangs meistens von großem Nachteil ist viele Steine zu besitzen, ist es am Ende dennoch der spielentscheidende Faktor. Deshalb fließt in die Bewertungsfunktion auch die Anzahl der Steine ein, die jeder Spieler zum Zeitpunkt der Evaluation auf dem Spielfeld hat.

Algorithm 7 coinsEval

```
1: maxCoins = getNumberOfCoinsCaptured(turnplayer)
2: minCoins = getNumberOfCoinsCaptured(oppositePlayer)
3:
4: return 100*(maxCoins - minCoins)/(maxCoins + minCoins)
```

Gewichtung

Die einzelnen Faktoren werden im Nachhinein noch einzeln gewichtet, um eine optimale Bewertung der Position zurückzugeben. Zudem lassen sich die einzelnen Konfigurationen der Gewichtungen in Abhängigkeit des Spielfortschritts verändern. Das wird mithilfe folgender Funktion gemacht:

$$\text{turnDependend\_Weight} = \frac{\text{turn\_Factor}}{60 \cdot \text{current\_Turn}} + \text{Weight} - \frac{1}{2} \cdot \text{turn\_Factor}\,.$$

Wenn zum Beispiel der Faktor der „Anzahl der Steine“ am Anfang eine kleine Rolle spielt aber dessen Relevanz gegen Ende zunimmt, dann kann dessen Gewichtung ganz einfach im Verlaufe des Spiels

Algorithm 8 dynamicEval

```
1:  eval = 0
2:
3:  if staticEvalWeight > 0 then
4:      eval += turnDependend_StaticEvalWeight * staticEval()
5:  else if flanksEvalWeight > 0 then
6:      eval += turnDependend_flanksEvalWeight * flanksEval()
7:  else if coinsEvalWeight > 0 then
8:      eval += turnDependend_coinsEvalWeight * coinsEval()
9:  else if mobilityEvalWeight > 0 then
10:     eval += turnDependend_mobilityEvalWeight * mobilityEval()
11: else if stabilityEvalWeight > 0 then
12:     eval += turnDependend_stabilityEvalWeight * stabilityEval()
13: else if xcfieldsEvalWeight > 0 then
14:     eval += turnDependend_xcfieldsEvalWeight * xcfieldsEval()
15: else if cornersEvalWeight > 0 then
16:     eval += turnDependend_cornersEvalWeight * cornersEval()
17: end if
```

linear erhöht werden.
Wenn sich ein Spielfeld im Endzustand befindet, dann wird nicht die Bewertungsfunktion angewandt, sondern durch das Zählen der Steine ermittelt, welcher Spieler gewonnen hat und dementsprechend $Integer.MAX\_VALUE$ oder $Integer.MIN\_VALUE$ zurückgegeben.

4 Monte Carlo Tree Search

Der Algorithmus bekannt als Monte Carlo Tree Search (MCTS) wurde 2006 von dem französischen Informatiker RÉMI COULOM als Algorithmus zum Spiele von Go vorgeschlagen[18]. MCTS ist ein probabilistischer Suchalgorithmus, der auf der Monte Carlo Methode basiert. Bei der Monte Carlo Methode wird mit Hilfe von wiederholten zufälligen Simulationen versucht ein deterministisches Problem zu lösen.[9]

4.1 Algorithmus

Da MCTS versucht den Teilbaum des vielversprechendsten Zuges am weitesten zu erforschen, muss der Baum konstant aktualisiert werden. Daher wird der Algorithmus in 4 Schritte aufgeteilt:

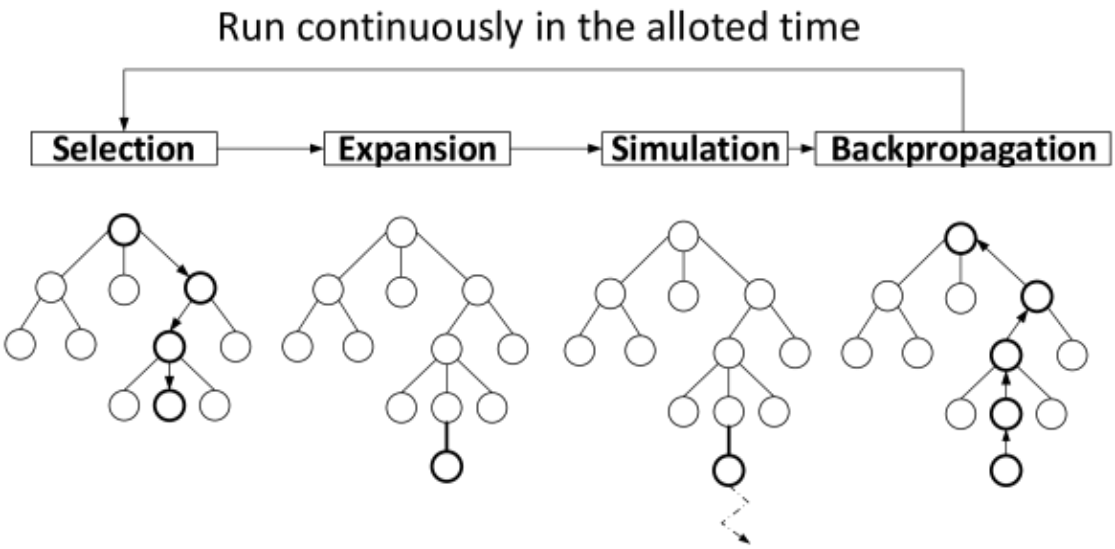

Abbildung 4.1: MCTS Lebenszyklus[18]

Selection Im Selektionsschritt wird der Baum, im Wurzelknoten anfangend, durchlaufen, bis ein Blattknoten erreicht wird. Dabei wird immer der Kindknoten mit der höchsten Gewinnwahrscheinlichkeit gewählt oder wenn eine UCT-Funktion genutzt wird, der Kindknoten, der die diese maximiert.

Wenn einer der untergeordneten Knoten noch nicht expandiert wurde, dann wird die Erforschung dieses Knotens immer bevorzugt. und das Ergebnis der UCT-Funktion ignoriert.

Expansion Wenn der Spielzustand des zu expandierenden Knotens kein Endzustand ist, dann werden alle möglichen Züge und die dazugehörigen Spielstände generiert. Daraufhin wird auf einem der Kindknoten der nächste Schritt, nämlich die Simulation angewandt. Wenn die Selektion zu einem Knoten führt, der noch nicht simuliert wurde, dann wird dieser erst simuliert und die Expanion übersprungen.

Simulation Im Simulationsschritt, auch bekannt als „rollout“, wird auf dem Spielzustand des im Selektionsschritt gewählten Knotens, eine Monte Carlo Simulation ausgeführt. Dabei wird das Spiel von diesem Zustand aus mit zufällig generierten Zügen zu Ende gespielt. Einem Sieg kann dann zum Beispiel der Wert 1, einem Unentschieden der Wert 0 und einer Niederlage der Wert -1 zugewiesen werden. Das Ergebnis der Simulation wird in dem Knoten festgehalten.

Backpropagation Im letzten Schritt wird der aus der Simulation ausgerechnete Wert nach und nach auf die Bewertungen aller Elternknoten bis zum Wurzelknoten addiert. Dabei ist zu beachten, dass bei Knoten, bei denen nicht der gleiche Spieler am Zug ist, wie im gerade simulierten Knoten, die Wertung vor der Addition zu negieren ist.
Zusätzlich wird bei allen durchlaufenen Elternknoten ein Zähler erhöht, der mitverfolgt, wie oft ein Knoten besucht wurde.

4.2 UCT

Die Auswahl der richtigen Züge in der Selektionsphase ist ein wichtiges Thema bei MCTS. Konzentriert sich der Algorithmus auf die Exploration des Suchbaumes, dann sucht dieser zu sehr in die Breite und konvergiert zu einem ineffizienteren Minimax Algorithmus. Konzentriert sich der Algorithmus auf die sogenannte Ausbeutung guter Züge, dann sucht er zu sehr in die Tiefe. Es muss also ein gesundes Gleichgewicht zwischen Exploration und Ausbeutung gefunden werden.
Die UCT-Formel (Upper Confidence Bound 1 applied to trees) ist eine Funktion, mit der man ein Gleichgewicht zwischen Exploration und Ausbeutung finden kann und ist folgendermaßen formuliert[9]:

$$\mathrm{UCT} = \frac{w_i}{n_i} + c\sqrt{\frac{\ln\left(N_i\right)}{n_i}}\,.$$

- w_i Anzahl der Siege des i-ten Knotens
- n_i Anzahl der Male, die der i-te Knoten besucht wurde
- N_i Anzahl der Male, die der Elternknoten des i-ten Knotens besucht wurde
- c Explorationsfaktor, mit dem das Verhältnis zwischen Exploration und Ausbeutung abgepasst werden kann

Der linke Summand der UCT-Formel stellt die Ausbeutung und der rechte Summand die Exploration dar. Je größer c gesetzt wird, desto mehr wird die Exploration gegenüber der Ausbeutung bevorzugt. Theoretisch liegt das Optimum bei $\sqrt{2}$, solange sich die Evaluationswerte zwischen 0 und 1 befinden.[9] Dieser muss in der Realität aber empirisch für jeden Fall einzeln bestimmt werden.
Man kann den MCTS-Algorithmus mithilfe einer simplen Evaluation gerichtet explorieren lassen, indem man die UCT-Funktion um einen weiteren Summanden erweitert, auch progressive bias genannt[9]:

$$\mathrm{UCT}_{\mathrm{directed}} = \frac{b_i}{n_i} + \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}}$$

- b_i Heuristische Bewertung des i-ten Knotens

Da die Evaluation für einen Knoten immer denselben Wert berechnet, nimmt die Gewichtung dieses Summanden ab, je öfter der Knoten besucht wird. Er dient also lediglich dazu bei der anfänglichen Exploration eines Knotens Züge zu erst zu prüfen, die vermutlich gut sind.

4.3 Bekannte Vor- und Nachteile von MCTS

MCTS hat gegenüber klassischen Spielbaum-Suchverfahren, wie zum Beispiel Alpha-Beta, 2 Vorteile. Erstens benötigt es keine Bewertungsfunktion und somit kein Wissen über Strategien und Taktiken im Spiel, um korrekt implementiert zu werden. Zweitens wird der Suchbaum nicht gleichmäßig durchsucht, sondern stärker in Regionen, die aussichtsreicher sind. Er wächst somit asymmetrisch.[9] Damit kann auch eine höhere Suchtiefe in diesen Teilbäumen erreicht werden.

Es hat aber auch zwei Nachteile: MCTS tut sich mit Spielständen schwer, die schlecht aussehen, wo aber eine Abfolge einiger präziser Züge zu einer guten Position oder auch einem Sieg führen kann und umgekehrt. Hinzu kommt, dass MCTS, Pfade, die zu einem *Sudden Win* oder einem *Sudden Lose* führen nicht wirklich erkennt und zwischenspeichert, weshalb es im Endspiel relativ untauglich ist. MCTS kommt auch mit aufwendigen Simulationen nicht gut klar, da ihm dadurch eine kleinerer Suchbaum und weniger Stichproben zur Verfügung stehen.[9]

4.4 MCTS-Optimierungen

Auch für MCTS gibt es weitere Optimierungen, welche die Schwächen des Algorithmus' ausgleichen sollen.

Endspielsuche

Bei einem Brettspiel wie Othello, welches endet, sobald alle 64 Steine auf dem Feld liegen, kann man den MCTS um eine Endspielsuche erweitern. Gegen Ende des Spiels wird das MCTS-Suchverfahren durch eine Alpha-Beta Suche ersetzt, die ausschließlich beendete Spielstände bewertet und nach einem Sieg sucht.

Mehrere Simulationen pro Rollout

So eine Monte Carlo Simulation kann sehr zeitaufwendig sein und das Ergebnis nur einer Simulation ist nicht wirklich aussagekräftig. Aus diesem Grund könnte man an Stelle einer einzigen Simulation, mehrere Simulationen ausführen und dem Knoten den Durchschnitt aller Ergebnisse, also so etwas wie eine Gewinnwahrscheinlichkeit, zuweisen. Die Ausführung dieser könnte man entweder sequentiell oder auch parallel ausführen. Durch Parallelisierung müssten im selben Zeitraum mehr Simulationen durchgeführt und im Optimalfall gleich viele Knoten durchsucht werden.

4.5 Experiment MCTS mit Bewertungsfunktion

Monte Carlo Simulationen können an sich relativ aufwendig sein. Das trifft vor Allem bei Brettspielen wie Othello zu, wo das Durchführen eines Zuges aus mehr als nur dem Bewegen oder Setzen einer Figur besteht. Aus diesem Grund wurde hier ein weiterer Ansatz gewählt, bei dem man die Monte Carlo Simulation in dem MCTS ganz einfach durch die Bewertungsfunktion des Alpha-Beta Algorithmus‘ ersetzt.
Dieser neue Algorithmus ist in der Theorie deutlich schneller als MCTS, erreicht größere Tiefen als Alpha-Beta und wenn die Bewertungsfunktion gut genug ist, dann sollte dieser auch die anderen Algorithmen schlagen können. Er kombiniert nämlich das Wissen und die Spielweise der Bewertungsfunktion von Minimax mit der dynamischen asymmetrischen Exploration des Spielbaumes von MCTS. Dabei wird die nicht gerichtete UCT-Funktion für die Selektion benutzt. Die ausgerechnete Evaluation wird durch 100 geteilt, um Werte zwischen -7 und 7 zu erhalten und ein Sieg mit 5 Punkten bewertet. Eine Niederlage wird äquivalent dazu mit -5 bewertet. Im Folgenden wird die Implementation dieses Algorithmus auch „MCTS EVAL“ genannt.

5 Implementierungen

Die Algorithmen wurden so implementiert, dass man die Möglichkeit hat einzelne Optimierungen wie Iterative Deepening oder Transposition Tables nach belieben, vor dem Start des Spiels, ein- und auszuschalten. Dafür können die BOOLEAN-Variablen $TRANSPOSITION\_TABLE$ und $ITERATIVE\_DEEPENING$ verändert werden. Die Zugring-Optimierung nutzt andere Funktionen und wird von einer anderen Wrapper-Klasse aufgerufen.

5.1 Alpha-Beta

In klassischen Minimax und Alpha-Beta Implementierungen werden die relevanten Daten als Parameter dem nächsten Rekursionsschritt übergeben. Da hier aber einzelne Optimierungen, wie Transposition Tables genutzt werden können, welche das zwischenspeichern dieser Informationen benötigen, werden die zum Spielfeld gehörenden Informationen in einem Java-Objekt gespeichert. Diese Informationen, sowie eine Hashfunktion dieser, werden in einem „State“ Objekt gespeichert.

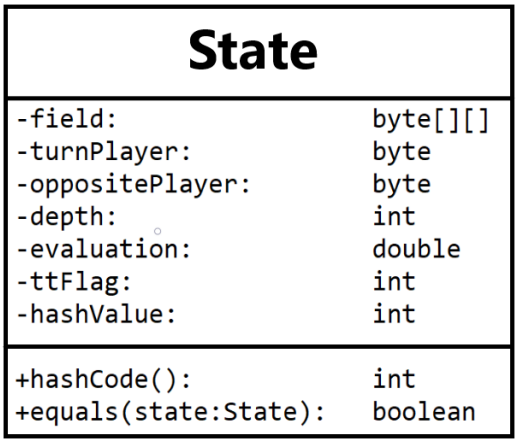

Abbildung 5.1: State Objekt

Algorithm 9 alphaBetaWrapper

```
depthToReach = 0
alpha = INT.MIN
beta = INT.MAX
passes = 0
while time left do alphaBeta(state,alpha,beta,passes)
    if not terminated then getBestMove() depthToReach++
    end if
end while
```

Algorithm 10 alphaBeta

```
if passes ≥ 2 then
    return determineWinner()
end if
if TRANSPOSITION_TABLE then
    findTransposition()
    if transposition found then
        performTranspositionCheck()
        if alpha ≤ beta then
            return tableEntry.eval
        end if
    end if
end if
if depthToReach was reached then
    eval = dynamicEval(field,turnPlayer)
    if ITERATIVE_DEEPENING then
        saveState()
    end if
    return eval
end if
findAllNextMoves()
createChildStates()
if ITERATIVE_DEEPENING then
    sortChildStates()
end if
max = alpha
```

```
26: if no next move possible then
27:     newState = createChildStateAfterPass()
28:     eval = -alphaBeta(newState,-beta,-max,passes+1)
29:     if eval > max then
30:         max = eval
31:     end if
32: else
33:     while not all cildStates checked do
34:         newState = snatchFirstChildState()
35:         eval = -alphaBeta(newState,-beta,-max,passes)
36:         if eval > max then
37:             max = eval
38:             if currentDepth is 1 then
39:                 bestMove = newState.move
40:             end if
41:             if max >= beta then
42:                 break
43:             end if
44:         end if
45:     end while
46: end if
47: if TRANSPOSITION_TABLE then
48:     getTranspositionFlag()
49:     saveTransposition()
50: end if
51: return max
```

5.2 MCTS

MCTS wurde ähnlich wie Alpha-Beta mit der Möglichkeit verschiedene Optimierungen und Varianten nach Belieben ein- und auszuschalten, programmiert, um auch hier einfacher testen zu können, welche der Optimierungen tatsächlich etwas bringt und wie groß die Auswirkung jedes Parameters ist.

Algorithm 11 mcts

```
1: while time left do
2:     while currentNode isn't leafNode do
3:         if childNodes not created then
4:             expand(currentNode)
5:         end if
6:         if currentNode has no Children then
7:             createChildAfterPass()
8:         end if
9:         currentNode = getChildThatMaximizesUCT()
10:    end while
11:    rollout()
12:    backpropagate(currentNode,currentNode.sore)
13: end while
```

Algorithm 12 backpropagation

```
1: while root not reached do
2:     currentNode.timesVisited++
3:     currentNode.score += sore
4:     currentNode = correntNode.parent
5: end while
```

Algorithm 13 rollout

```
1: while game not over do
2:     playRandomMove()
3: end while
4: if win then
5:     return 1
6: else if lose then
7:     return -1
8: else
9:     return 0
10: end if
```

Wenn nur noch 12 Steine zu spielen sind, dann wird MCTS durch die Endspielsuche ersetzt. Aus ersten Tests hat sich ergeben, dass Alpha-Beta im Endspiel problemlos die 12te Stufe durchsuchen kann.

Den MCTS mit Bewertungsfunktion Ansatz aus Abschnitt 3.3 wurde wie MCTS implementiert. Der einzige Unterschied ist die Rollout-Funktion. Diese gibt einen durch die dynamische Evaluation ausgerechneten Wert, an Stelle des Ergebnisses der Simulation, zurück.

6 Tests

Für die Tests wurden zufällige Spielzustände nach 5, 10, 15, 20, 25 und 30 Zügen generiert un die Algorithmen darauf getestet.

6.1 Parameteroptimierung

Um eine Konfiguration von Gewichtungen für die einzelnen Parameter der Bewertungsfunktion zu finden, die gut genug war um die anderen Konfigurationen regelmäßig zu schlagen, wurden zuerst manuell Gewichtungen für die einzelnen Bewertungsfaktoren eingegeben. Dann wurden, mithilfe eines Skripts, einzelne Gewichtungen inkrementell angepasst und die Alpha-Beta Algorithmen gegeneinander spielen gelassen. Folgende Konfiguration hat die meisten Siege erzielt:

- $CORNERS\_FACTOR = 1$
- $MOBILITY\_FACTOR = 1$
- $X\_C\_FIELDS\_FACTOR = 1.5$
- $COIN\_QTY\_FACTOR = 0.01$
- $STABLE\_DISC\_FACTOR = 0$
- $STATIC\_EVAL\_FACTOR = 2$
- $FLANK\_FACTOR = 1$

Im nächsten Schritt wurde die Abhängigkeit der einzelnen Parameter von dem Fortschritt des Spiels getestet und folgendes Ergebnis ist dabei herausgekommen:

- $CORNERS\_TURN\_FACTOR = 1$
- $MOBILITY\_TURN\_FACTOR = 1$
- $X\_C\_FIELDS\_TURN\_FACTOR = 1.5$
- $COIN\_QTY\_TURN\_FACTOR = 0$
- $STABLE\_DISC\_TURN\_FACTOR = 0$
- $STATIC\_EVAL\_TURN\_FACTOR = 0$
- $FLANK\_TURN\_FACTOR = -1$

Diese Konfiguration hat gegen die vorherige, welche nicht von dem Spielfortschritt abhängig ist, in rund $69\,\%$ der Spiele geschlagen.

6.2 Alpha-Beta

Um die Performanz der einzelnen Optimierungen zu vergleichen haben die Algorithmen bei verschiedenen Zeitlimits auf zufällig generierten Spielzuständen den besten Zug ausgerechnet. Dabei wurden einzelne Optimierungen ein und ausgeschaltet und die Resultate separat festgehalten. Als Vergleichswert wurde die erreichte Suchtiefe genommen. Die Ergebnisse wurden in folgenden Grafiken festgehalten:

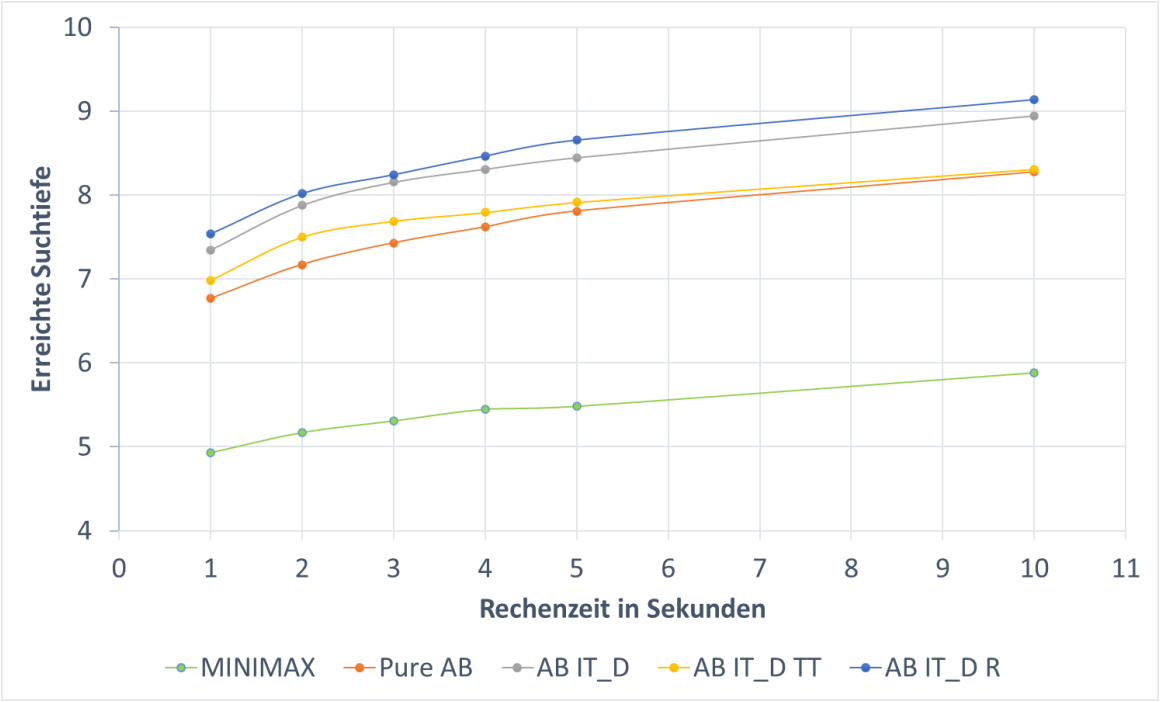

Abbildung 6.1: Minimax Optimierungen erreichte Suchtiefe

Der Minimax Algorithmus durchsucht im Schnitt $44,16\,\%$ mehr Knoten als die Variante mit Alpha-Beta Pruning. Dafür kommt Alpha-Beta in den Testdaten auf eine durchschnittliche Suchtiefe von $7,51$, das ist $39,85\,\%$ tiefer als Minimax. Diesen $Trade-Off$ lohnt es sich einzugehen, da bei Algorithmen wie diesen, für das Spielen des Spiels, allein die erreichte Suchtiefe relevant ist.

Iterative Deepening mit Vorsortierung der Züge hat die durchschnittlich erreichte Suchtiefe weiter noch um $8,39\,\%$ auf $8,14$ erhöht,

Das Nutzen von Transpotition Tables verringert die durchschnittlich erreichte Suchtiefe wieder. Bei 10 Sekunden pro Spielzug wird der fast der komplette Nutzen, den Iterative Deepening gebracht hat wieder annulliert und die Variante erreicht eine Suchtiefe wie Alpha-beta ohne Zugvorsortierung.

Der Zugring führt zu einer leichten Verbesserrung der Suchtiefe um $1,23\,\%$.

6.3 MCTS

Beim Vergleich der Alpha-Beta Algorithmen war klar: die Konfiguration, welche die größte Tiefe erreicht, ist auch die beste, da alle dieselbe Bewertungsfunktion benutzen. Da bei MCTS die UCT Funktion die Suchtiefe beeinflusst, macht es hier mehr Sinn die Anzahl durchsuchter Spielzustände und die Gewinnrate als Vergleichswerte zu nehmen, was zu einem größeren Simulationsaufwand führt, da ganze Spiele zwischen Algorithmen simuliert werden müssen:

Durchsuchte Spielstände

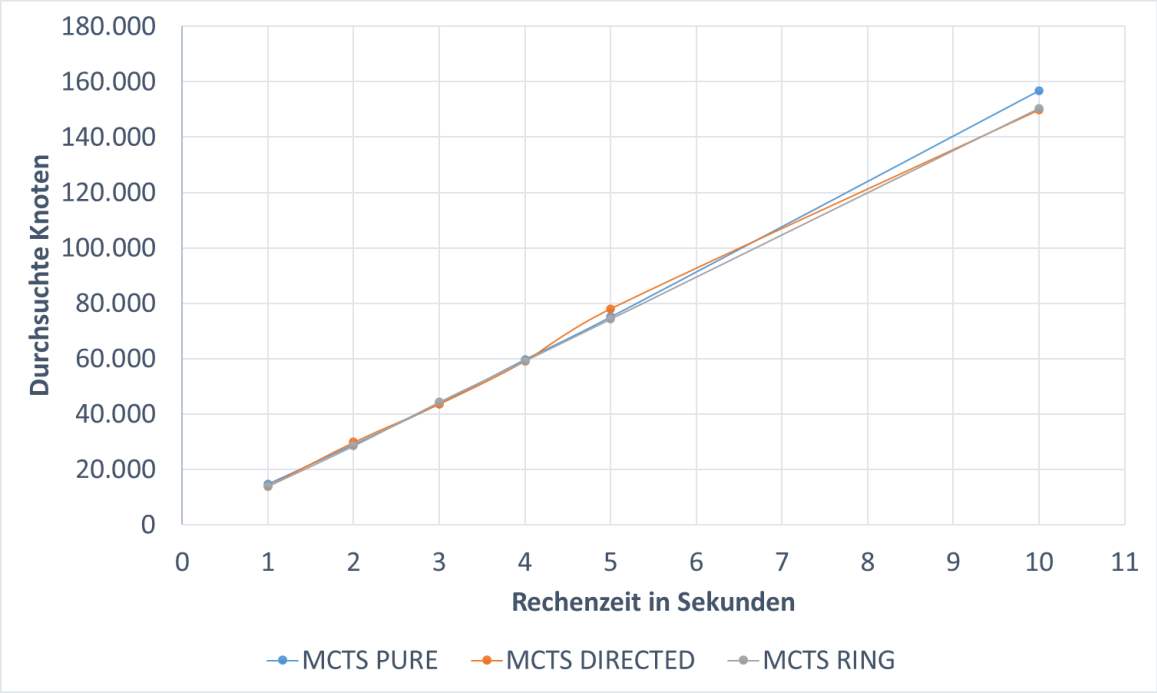

Abbildung 6.2: MCTS Optimierungen erreichte Spielstände bei 1 Simulation pro Rollout

Das einbinden des Evaluationssummanden in die UCT-Funktion ist mit sehr geringem Rechenaufwand verbunden. Die Anzahl durchsuchter Knoten ist nur um $1,2\,\%$ gesunken.
Wie man aus der Abbildung noch erkennen kann, verringert die Nutzung des Zugrings, anders als bei Alpha-Beta, hier Performanz des Algorithmus's. Sie bringt hier also keine Vorteile.

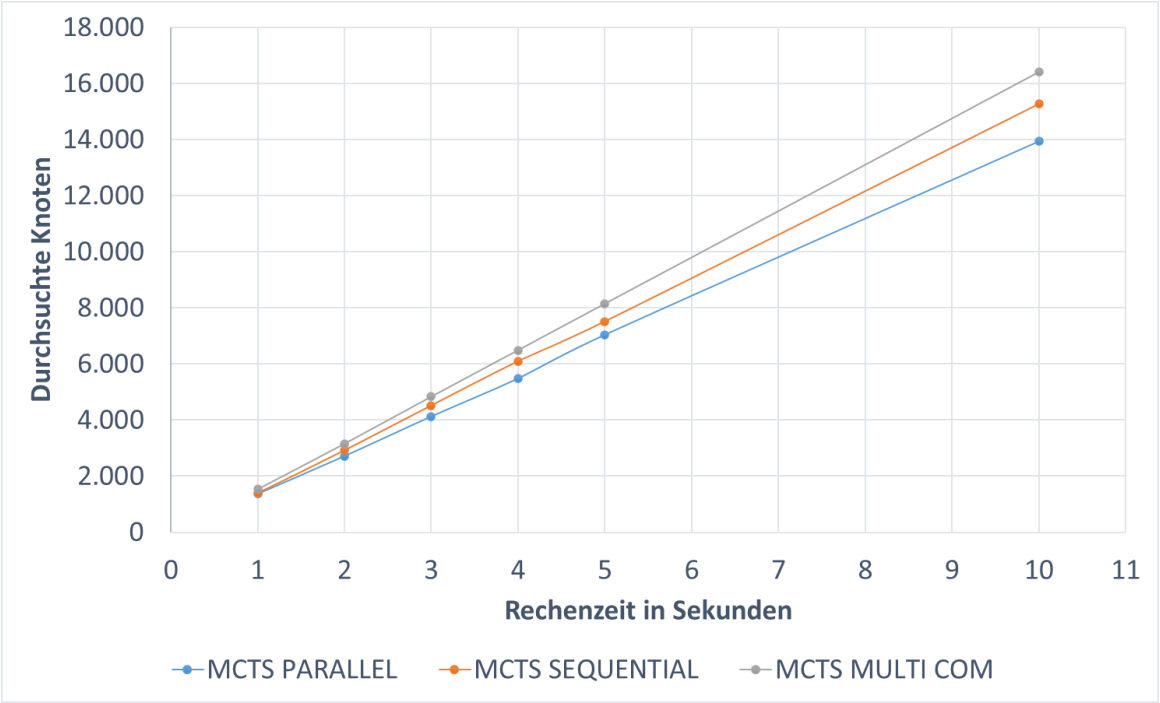

Abbildung 6.3: MCTS Optimierungen erreichte Spielstände bei 10 Simulationen pro Rollout

Wie man erkennt werden bei 10 Simulationen pro rollout deutlich weniger Knoten durchsucht. Das liegt daran, dass die Erstellung eines Thread-Objektes in Java etwas Zeit benötigt. Wenn man zum Beispiel an Stelle von 10 Thread-Objekten mit jeweils einer Simulation nur 1 Thread-Objekt mit 1 Simulation erstellt und den Algorithmus laufen lässt, dann werden nach 1 Sekunde Rechenzeit im Schnitt nur 4.379 Knoten bewertet, im Vergleich zu den 14.691. Bei 10 gleichzeitig rechnenden Threads werden 1.369 Knoten durchsucht, was 13.690 Simulationen entspricht. Das sind nur $8,4\,\%$ Simulationen weniger, als bei MCTS PURE.
Weiterhin fällt auf, dass das sequentielle Ablaufen lassen der Simulationen bessere Ergebnisse erzielt. Das liegt daran, dass die Simulationen im Verlaufe des Spiels kürzer werden und sich die Parallelisierung nicht mehr lohnt. Nach durchschnittlich 20 Zügen findet ein Umschwung statt und der sequentielle Ansatz liefert bessere Ergebnisse als der parallele.
Daraufhin wurden beide Algorithmen so kombiniert, dass nach 20 Zügen die sequentielle Simulation übernimmt. Der daraus entstehende Algorithmus MCTS MULTI COM schafft es $7\,\%$ mehr Simulationen durchzuführen als MCTS PURE.

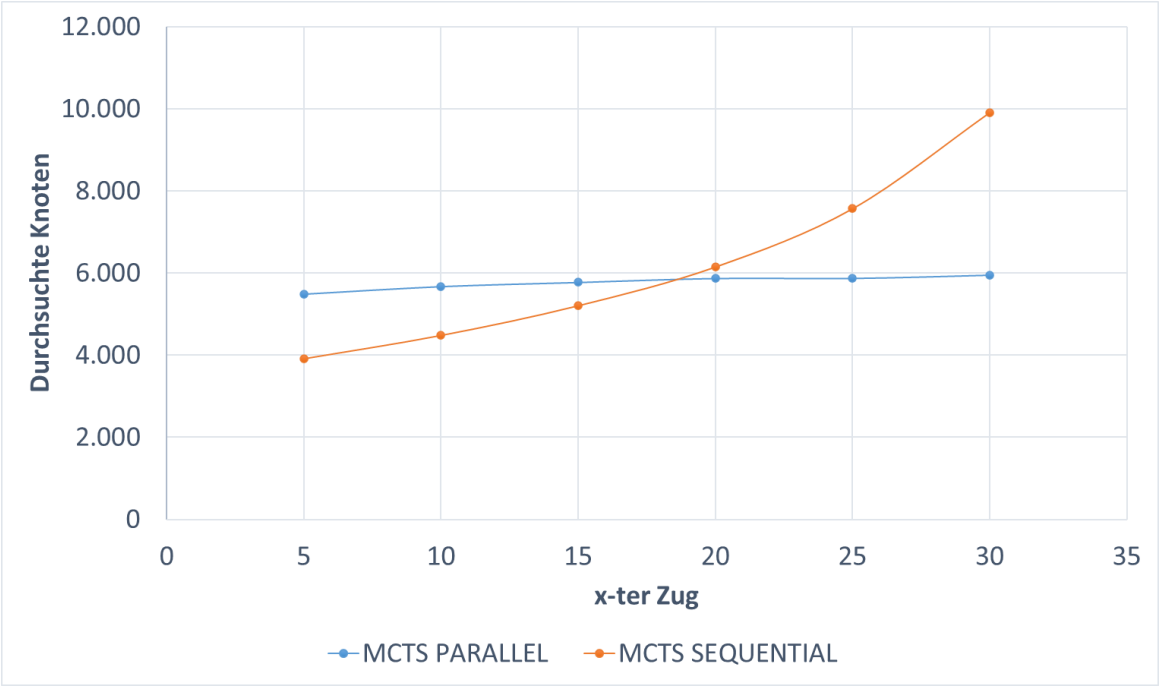

Abbildung 6.4: MCTS 10 Rollouts SEQ vs PARALLEL auf Spielständen nach x Zügen

Das Ersetzen der Monte Carlo Simulation durch eine Bewertungsfunktion führt durch die geringeren Berechnungskosten, wie erwartet, auch zu einer deutlich größeren Anzahl an durchsuchter Knoten. Und zwar werden durchschnittlich 573.556 Simulationen durchgeführt. Das sind etwa 9 mal mehr Spielzustände als beim klassischen MCTS. Wie schon beim klassischen MCTS, verschlechtert auch hier der Zugring die Perfomanz.

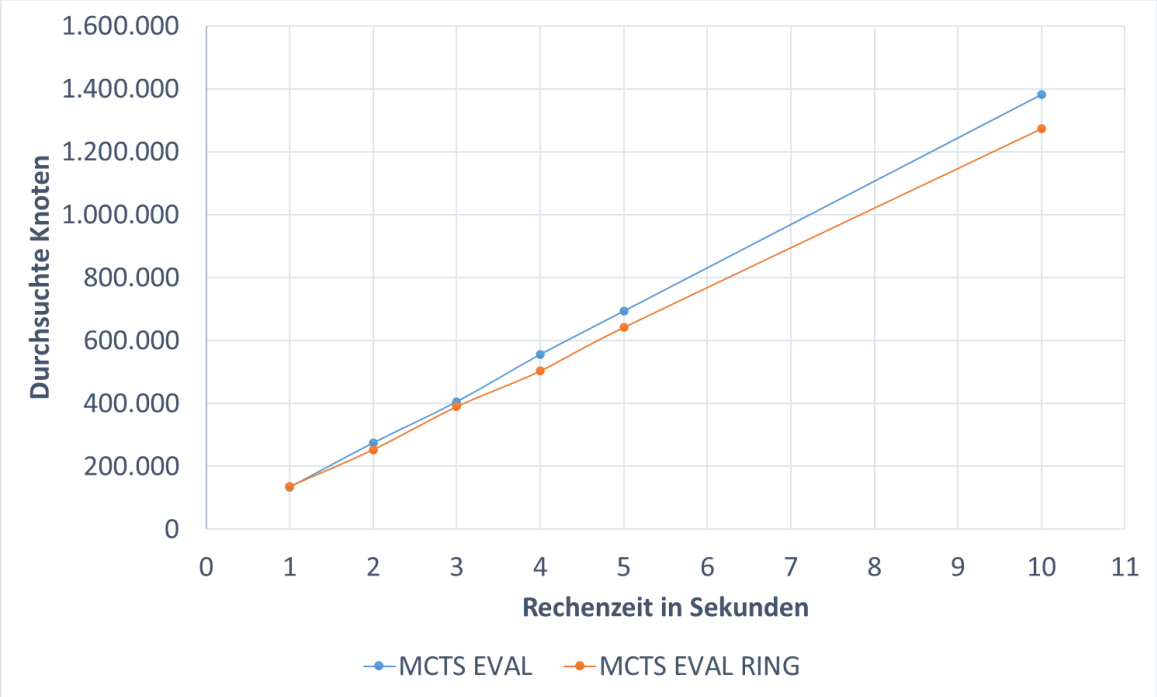

Abbildung 6.5: MCTS mit Bewertungsfunktion + Optimierung

MCTS Variante Gewinnrate

	MCTS PURE	MCTS DIRECTED	MCTS PARALLEL	MCTS MULTI COM
MCTS PURE	x	0,17	0,39	0,39
MCTS DIRECTED	0,83	x	0,39	0,72
MCTS PARALLEL	0,61	0,61	x	0,61
MCTS MULTI COM	0,61	0,28	0,39	x
Durchschnitt	0,68	0,35	0,39	0,57

Tabelle 6.1: MCTS und Optimierungen im Vergleich: Gewinnraten

Aus der Tabelle erkennt man ganz klar, dass MCTS PURE die beste Variante ist, mit einer durchschnittlichen Gewinnrate von $68,33\,\%$.MCTS MUTLI COM kommt ihr relativ nah mit einer Gewinnwahrscheinlichkeit von $57,33\,\%$. MCTS PARALLEL schneidet am schlechtesten ab, vermutlich wegen der Performanz der Thread-Objekte. Überraschenderweise schneidet auch MCTS DIRECTED unterdurchschnittlich ab.

6.4 Algorithmen im Vergleich

In diesem Abschnitt wurden die 3 besten Algorithmen aus den Kategorien „MiniMax“, „MCTS “ und „MCTS mit Bewertungsfunktion“ bei verschiedenen Zeitlimits und Startpositionen gegeneinander spielen gelassen, um herauszufinden, welcher der Algorithmen denn tatsächlich am Besten spielt:

Monte Carlo Tree Search vs andere Algorithmen

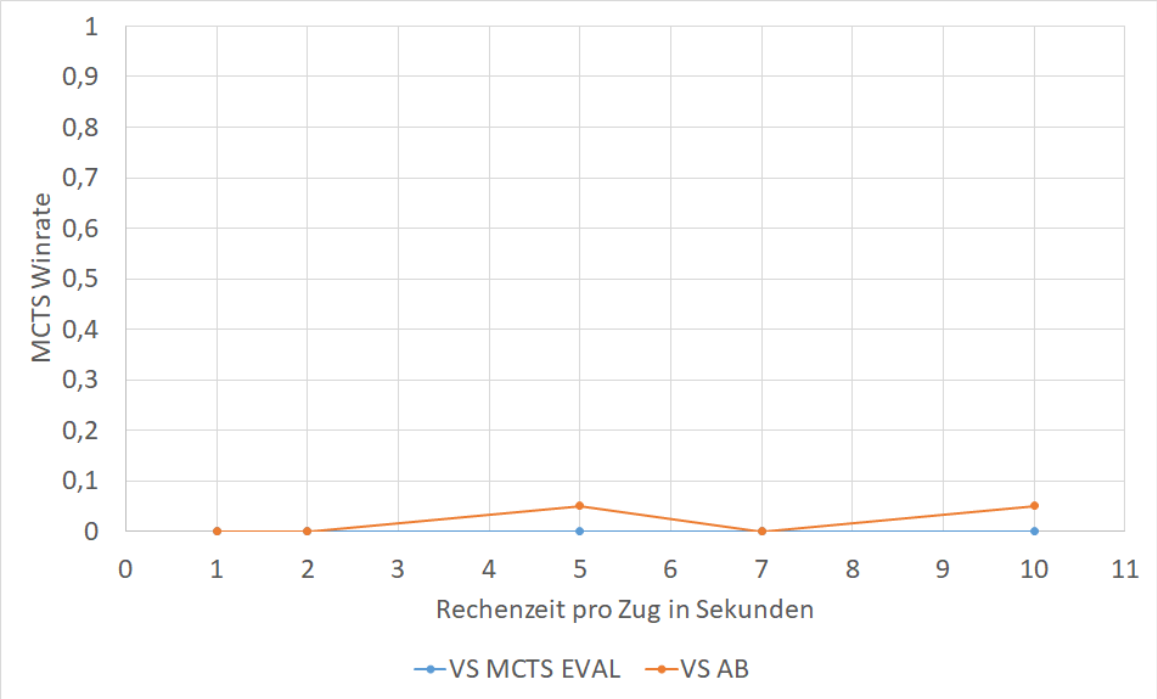

Abbildung 6.6: MCTS vs. Alpha-Beta/MCTS EVAL

Der klassische Monte Carlo Tree Search schneidet beim tatsächlichen Spielen von Othello sehr schlecht ab, mit einer Gewinnrate von 0 gegen MCTS mit Bewertungsfunktion und fast 0 gegen Alpha-Beta.
Wenn die Rechenzeit der anderen beiden Algorithmen nun exponentiell reduziert wird und dem MCTS 1s Rechenzeit zur Verfügung steht, dann wird folgendes Ergebnis erzielt:

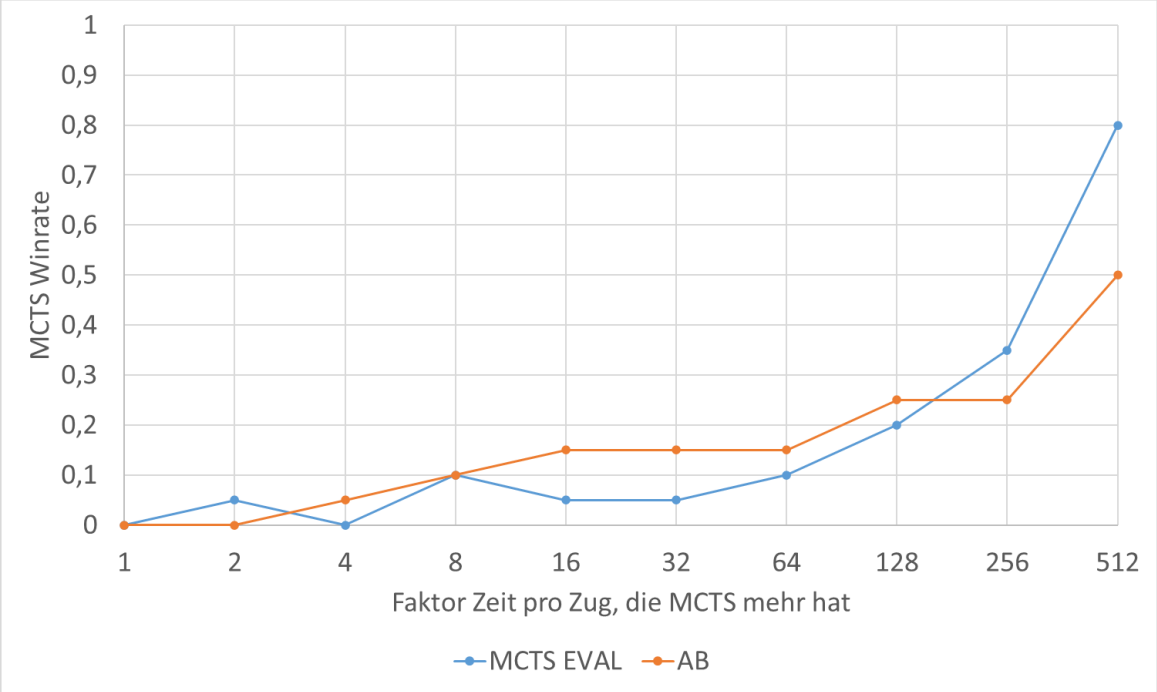

Abbildung 6.7: MCTS mit exponentiell mehr Zeit

Die Gewinnrate von MCTS nimmt sehr langsam zu. Bei 512 Mal mehr Zeit für MCTS, das entspricht ungefähr 2 ms Rechenzeit für die anderen beiden, fängt MCTS an die beiden Algorithmen regelmäßig zu schlagen.

Alpha-Beta vs. MCTS mit Bewertungsfunktion

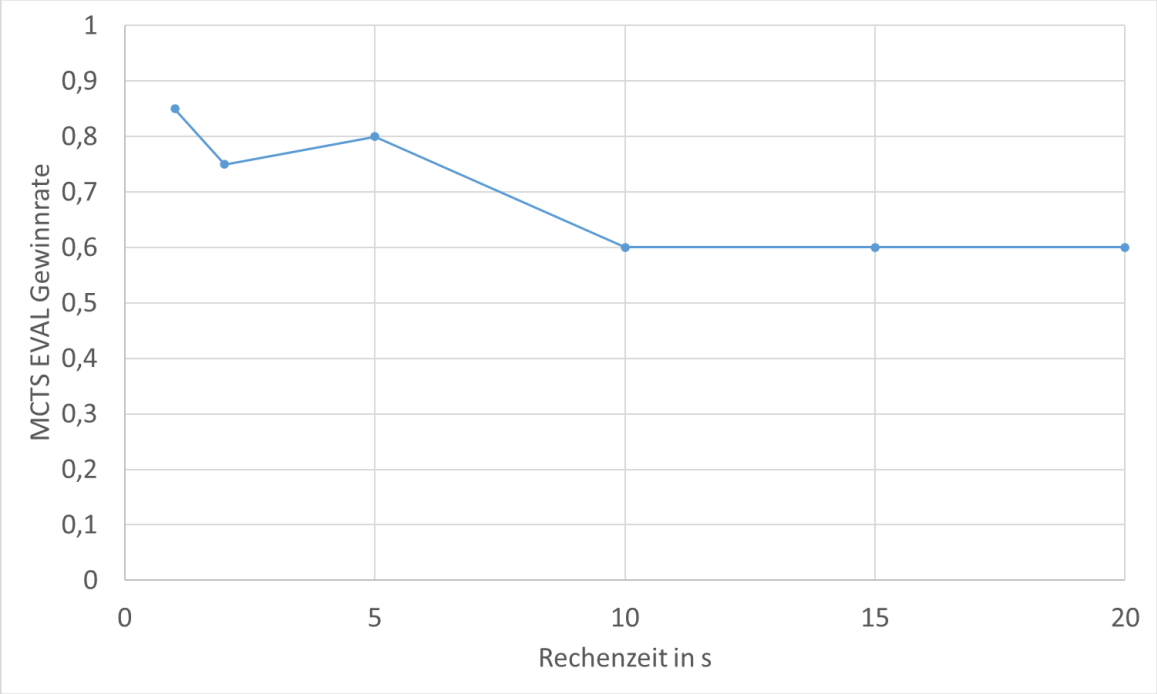

Abbildung 6.8: MCTS mit Bewertungsfunktion vs Alpha-Beta

MCTS EVAL mit einem Explorationsfaktor von $1,5$ schneidet besser als die beste Alpha-Beta Konfiguration ab. Vor allem bei Spielen mit relativ wenig Zeit pro Zug erreicht MCTS EVAL eine Gewinnquote von durchschnittlich $80\,\%$. Bei 10 Sekunden Zeit pro Zug sinkt die Gewinnrate auf $60\,\%$ und bleibt auch konstant dabei.
Da stellt sich natürlich die Frage ob die Anpassung des Explorationsfaktors bei mehr Rechenzeit die Gewinnrate erhöht. Weitere Tests zeigen jedoch, dass die Veränderung des Explorationsfaktors keinen großen positiven Einfluss auf die Gewinnquote haben.

7 Diskussion

Parameteroptimierung Die Berechnung von $STABLE\_DISC\_FACTOR$ ist zu aufwendig.Dieser Faktor hat im Vergleich zum Zeitaufwand einen zu geringen Einfluss auf das Endergebnis und anstatt diesen zu berechnen ist es sinnvoller mehr Knoten zu durchsuchen. Der $COIN\_QTY\_FACTOR$ hat einen vergleichweise geringen Einfluss auf die Bewertungsfunktion, ist aber dennoch zum Gewinnen notwendig. Der $FLANK\_FACTOR$ der einzige Faktor, dessen Einfluss mit steigender Rundenzahl abnimmt.

Alpha-Beta Das Nutzen von Transposition Tables die erreichte Suchtiefe verschlechtert. Die Anzahl gefundener Transpositionen beträgt auch nur $1,07\,\%$ der tatsächlich durchsuchten Spielzustände. Der zusätzliche Rechenaufwand ist also nicht lohnenswert. Das liegt daran, dass die in der Ausführung komplexeren Züge zu vielen verschiedenen Spielzuständen führen, die sich nicht so oft wiederholen. Die beste Variante hierfür ist also Minimax mit Alpha-Beta Pruning, Iterative Deepening und dem Zugring.

MCTS Die Nutzung des Zugringes führt, anders als bei Alpha-Beta, zu einer Verschlechterung der Performanz. Das liegt daran, dass das Durchsuchen des Spielbaums bei MCTS nur in eine Richtung geschieht. Der Zugring wird nach jedem Backpropagation-Schritt zurückgesetzt. Die Algorithmen können also gar nicht vollstens von dem Ring profitieren und das Zurücksetzen des Ringes ist aufwendiger, als der Performance-Boost den dieser bringt.

Monte Carlo Tree Search vs andere Algorithmen Das der klassische Monte Carlo Tree Search so schlecht abschneidet ist nicht überraschend. Dieser hat bereits in den vorherigen Tests deutlich weniger geleistet. Das liegt an den viel aufwendigeren Spielzügen und der daraus resultierenden teuren Simulation. Die Menge an Rechenzeit, die MCTS benötigt, um besser als die anderen Algorithmen zu spielen ist dennoch größer als erwartet.

Alpha-Beta vs. MCTS mit Bewertungsfunktion MCTS mit Bewertungsfunktion ist also im Spiel Othello besser als Alpha-Beta mit iterativer Tiefensuche und dem Zugring. Das heißt dass es besser ist Teile des Baumes einfach zu ignorieren und stattdessen eine größere Suchtiefe in den vielversprechenden Zweigen des Suchbaums zu erreichen.

8 Fazit

Zusammenfassend kann man sagen, dass sich der klassische Monte Carlo Tree Search Algorihmus für Spiele mit aufwendigen Zügen nicht gut eignet. Selbst die vorgestellte gerichtete Vaiante und die Variante mit mehreren Simulationen pro Knoten haben keine Chance gegen Alpha-Beta.
Beim Alpha-Beta Algorithmus macht es keinen Sinn Transposition Tables zu benutzen, wenn die Anzahl tatsächlicher Transpositionen im Suchbaum so gering wie bei Othello ist. Iterative Tiefensuche und der Zugring sind jedoch gerne gesehen und Alpha-Beta erreicht bei Othello mit diesen $11\,\%$ größere Tiefen als ohne.
MCTS mit Bewertungsfunktion ist wiederum ein wirklich guter alternativer Lösungsansatz. Die Verbindung aus dem Wissen der Bewertungsfunktion des Alpha-Beta Algorithmus' und der asymmetrischen Baumsuche des MCTS erzielt in dieser Arbeit das beste Ergebnis. Der Zugring ist auch hier wieder gutes ein Beispiel dafür, dass nicht jede Verbesserung, die in der Theorie gut ist, auf jeden Algorithmus zutrifft.
Das impliziert, dass es manchmal besser ist Genauigkeit für Suchtiefe aufzugeben. Das macht irgendwo auch Sinn, weil es in Brettspielen meistens reicht sehr gute Züge zu spielen, um zu gewinnen. Es muss nicht immer der beste Zug gefunden werden, solange der gewählte Zug auch zum Sieg führt.

Für die Zukunft wäre es interessant herauszufinden wie MCTS EVAL noch verbessert werden kann und ob es andere Spiele gibt, bei denen dieser Kombinierte Ansatz genutzt werden kann.

Literaturverzeichnis

[1] Alphago. https://www.deepmind.com/research/highlighted-research/alphago. Accessed: 19.10.2023.

[2] Chess programming part ii: Data structures. https://www.gamedev.net/tutorials/_/technical/artificial-intelligence/chess-programming-part-ii-data-structures-r1046/. Accessed: 19.10.2023.

[3] Introduction to stragies for othello. https://www.ultraboardgames.com/othello/strategy.php. Accessed: 19.10.2023.

[4] Othello. http://gamescrafters.berkeley.edu/games.php?game=othello. Accessed: 19.10.2023.

[5] Othello: Spielregeln. https://info.lite.games/de/support/solutions/articles/60000688960-othello-spielregeln. Accessed: 19.10.2023.

[6] Reversi and othello – two different games. do you know their different rules? https://bonaludo.com/2016/02/18/reversi-and-othello-two-different-games-do-you-know-their-different-rules/. Accessed: ?

[7] World othello championship. https://www.worldothello.org/about/tournaments/world-othello-championship. Accessed: ?

[8] Zugzwang. https://www.chess.com/de/terms/zugzwang. Accessed: ?

[9] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[10] Jill Cirasella and Danny Kopec. The history of computer games. 2006.

[11] Wolfgang Ertel and Nathanael T Black. *Grundkurs Künstliche Intelligenz*, volume 4. Springer, 2016.

[12] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43(1):21–36, 1990.

[13] John McCarthy, Marvin L. Minsky, Nathaniel Rochester, and Claude E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, 27(4):12, Dec. 2006.

[14] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann, 1992.

[15] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. An analysis of heuristics in othello, 2015.

[16] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, pages 124–130, 1996.

[17] Reinhard Selten. Die konzeptionellen grundlagen der spieltheorie einst und jetzt. Technical report, Bonn Econ Discussion Papers, 2001.

[18] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, jul 2022.