Bachelorarbeit

Vergleich der Suchalgorithmen Alpha-Beta, MTD(f) und Best-First-Minimax für das Othello-Spiel

Wissenschaftliche Arbeit zur Erlangung des akademischen Grades Bachelor of Science an der Fakultät für Elektrotechnik und Informatik der Technischen Universität Berlin

Semra Ayalp
MatrNr. 394746
Berlin, 9. November 2023

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

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

Kurzfassung

Suchalgorithmen spielen für die Entwicklung und Verbesserung von Spielen eine wichtige Rolle. Die Algorithmen *Alpha-Beta (AB)*, *Best-First-Minimax (BFM)* und *MTD(f)* haben sich in verschiedenen Spielen, insbesondere im Spiel *Othello*[1], als erfolgreich erwiesen, was in dieser vorliegenden Arbeit implementiert und als Testplattform verwendet wurde. Gleichzeitig spielen die Evaluationsmethoden eine bedeutende Rolle, da sie die Qualität der Entscheidungen, die durch die Suchalgorithmen getroffen werden, maßgeblich beeinflussen.

Das Ziel dieser wissenschaftlichen Arbeit besteht darin, eine gründliche Analyse und Bewertung dieser Suchalgorithmen und verschiedener Evaluationsmethoden durchzuführen, indem sie in einem selbst implementierten Othello-Spiel angewendet werden. Dafür werden die Suchalgorithmen und Evaluationsmethoden zunächst anhand von Pseudocodes aus verschiedenen wissenschaftlichen Arbeiten implementiert, um ihre Funktionsweise eingehend zu analysieren. Im Anschluss erfolgt ein detaillierter Vergleich dieser Suchalgorithmen und Evaluationsmethoden, bei dem die Ergebnisse in einer übersichtlichen und aussagekräftigen Form präsentiert werden. Dies ermöglicht nicht nur eine gründliche Bewertung ihrer Leistungsfähigkeit, sondern liefert auch wertvolle Erkenntnisse, die zur weiteren Verbesserung von Spielen beitragen können.

[1] Othello ist ein beliebtes Brettspiel, siehe Kapitel 3

Inhaltsverzeichnis

1. Einleitung

Suchalgorithmen spielen eine entscheidende Rolle bei der Entwicklung und Verbesserung von Spielen. Sie ermöglichen optimale Entscheidungen zu treffen und damit die Spiele zu verbessern. Insbesondere im Spiel *Othello* haben sich Algorithmen wie *Alpha-Beta (AB)*, *Best-First-Minimax (BFM)* und *MTD(f)* als äußerst erfolgreich erwiesen. In der vorliegenden Arbeit wurden diese implementiert und ihre Funktionsweise sowie Leistungsfähigkeit untersucht.

Neben den Suchalgorithmen sind auch Evaluationsmethoden von großer Bedeutung. Sie dienen dazu die Qualität der Entscheidungen, die durch die Suchalgorithmen getroffen werden, maßgeblich zu beeinflussen. Eine sorgfältige Auswahl und Bewertung der Evaluationsmethoden ist daher für die Optimierung von Spielen von entscheidender Bedeutung.

In der vorliegenden Arbeit liegt der Fokus auf der Analyse und Bewertung von Suchalgorithmen und Evaluationsmethoden anhand eines selbst implementierten Othello-Spiels. Dies erfolgt durch die Implementierung der Suchalgorithmen und Evaluationsmethoden basierend auf den Pseudocodes aus verschiedenen wissenschaftlichen Arbeiten. Anschließend werden sie auf ihre Funktionsweise hin eingehend analysiert und miteinander verglichen.

Abschließend werden die Ergebnisse in einer übersichtlichen Form präsentiert. Die gewonnenen Erkenntnisse können dazu beitragen, die Anwendung der Suchalgorithmen und Evaluationsmethoden in der Spielentwicklung zu verbessern und zukünftige Forschung in diesem Bereich anzustoßen.

2. Ähnliche Arbeiten

In der vorliegenden Arbeit werden verschiedene Suchalgorithmen und Evaluationsmethoden im Kontext des Othello-Spiels miteinander verglichen. Es existieren zahlreiche andere wissenschaftliche Arbeiten, die sich mit der Kombination verschiedener Suchalgorithmen und Evaluationsmethoden befassen und sie miteinander vergleichen.

Hernandez et al. haben in ihrer Arbeit die *Alpha-Beta-* und *Scout-Algorithmen* zur Optimierung des Othello-Spiels studiert und sie miteinander verglichen [6]. Alpha-Beta ist ein bekannter Minimax-Algorithmus mit einem sogenannten *Pruning*-Konzept, das die Anzahl der zu überprüfenden Knoten im Suchbaum reduziert. Im Gegensatz dazu stellt der Scout-Algorithmus eine Weiterentwicklung des Alpha-Beta-Algorithmus dar, bei dem lediglich vielversprechende Zweige des Spielbaums durchsucht werden. Die Autoren haben in ihrer Arbeit auch Evaluationsmethoden behandelt. Dabei wurden einige der Evaluationsmethoden wie *Statische Gewichte*, siehe Kapitel 4.1, *Parität*, siehe Kapitel 4.3 und *Mobilität*, siehe Kapitel 4.4, kurz beschrieben, die auch in dieser vorliegenden Arbeit zum Einsatz kamen. Die Ergebnisse der Studie zeigen, dass der Scout-Algorithmus eine überlegene Alternative zum Alpha-Beta-Algorithmus darstellt und in Bezug auf die benötigte Zeit eine bessere Leistung bietet.

Sannidhanam et al. widmeten sich in ihrer Arbeit den Suchalgorithmen *Minimax* und *Alpha-Beta* [15]. Sie führten eine detaillierte Analyse mehrerer Evaluationsmethoden durch, die ebenfalls in dieser vorliegenden Arbeit verwendet wurden. Ihre Studie führte schließlich zur Erkenntnis, dass die Evaluationsmethode *Ecken*, siehe Kapitel 4.2, die besten Ergebnisse lieferte. Es ist jedoch anzumerken, dass sie bei ihrem Vergleich die Evaluationsmethode mit statischen Gewichten, siehe Kapitel 4.1, außer Acht gelassen und sie nicht in die Bewertung einbezogen haben.

Tommy et al. haben in ihrer Arbeit die Algorithmen *Alpha-Beta* und *MTD(f)* in einem *Connect Four*-Spiel miteinander verglichen [17]. Dabei führten sie eine Analyse hinsichtlich des Gewinnprozentsatzes, der Ausführungszeit und der Anzahl der bearbeiteten Knoten durch. Die Ergebnisse zeigten, dass MTD(f) im Durchschnitt schneller als Alpha-Beta ist und weniger Knoten auswertet. Diese Ergebnisse wurden ebenfalls in dieser vorliegenden Arbeit bestätigt.

Barber et al. verglichen in ihrer Arbeit [3] die Algorithmen *MiniMax*, *Alpha-Beta* und *Scout* miteinander. Zusätzlich studierten sie verschiedene Evaluationsmethoden, unter anderem *Ecken (Corners Captured)*, *Mobilität*, *Anzahl von Steinen (Coin Parity)*, *Statische Gewichte (Static)*, und kamen zu dem Schluss, dass die Evaluationsmethode mit statischen Gewichten, siehe Kapitel 4.1, die besten Ergebnisse im Test erzielte, was auch in dieser Arbeit bestätigt wurde.

Korman präsentierte in seiner Arbeit verschiedene Evaluationsmethoden, die in einem Othello-Spiel verwendet werden können [9]. Dabei behandelte er auch Suchalgorithmen wie *Minimax* und *Alpha-Beta*.

3. Othello

3.1. Regeln

Othello ist ein beliebtes Brettspiel. Das Spiel wird mit zwei Spielern auf einem 8x8-Spielbrett gespielt. Spieler bekommen jeweils entweder die schwarze oder weiße Farbe. Das Spiel gewinnt der Spieler, der am Ende des Spiels mehr Steine auf dem Spielbrett hat. Nach Spielregeln muss das Spiel immer mit einer Startaufstellung starten, die in Abb. 3.1.*a* zu sehen ist [2], [3].

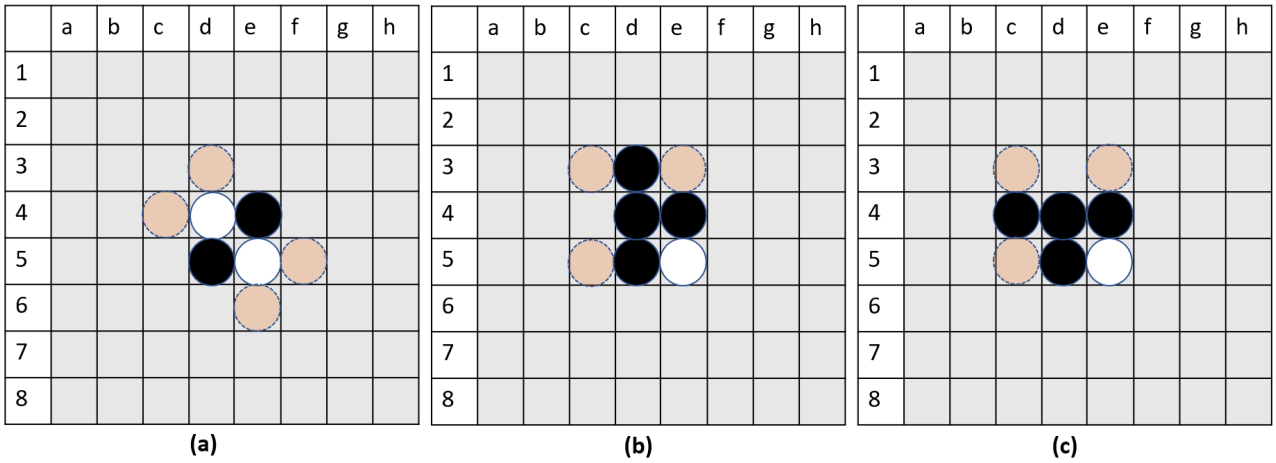


Abbildung 3.1.: (a) zeigt die Startaufstellung eines Othello-Spiels. Der schwarze Spieler beginnt das Spiel. Die orangen Kreise zeigen die möglichen Züge für den aktuellen Spieler. (b) und (c) zeigen den Stand des Spielbretts, nachdem der schwarze Spieler entweder auf $d3$ oder $c4$ gespielt hat. Anschließend ist der weiße Spieler an der Reihe.

Der schwarze Spieler beginnt das Spiel. Spieler setzen ihre Steine, oder auch *Disks* bzw. *Scheiben* genannt, abwechselnd auf das Spielbrett. Das gesetzte Feld muss vorher leer sein und mindestens einen gegnerischen Stein angrenzen. Nach einem Zug werden alle gegnerischen Steine, die horizontal, vertikal oder diagonal durch eigene Farben eingeschlossen werden können, in die Farbe des Spielers umgewandelt, wie es in Abb. 3.1.*b* bzw. 3.1.*c* zu sehen ist [7], [11].

Wenn ein Spieler an der Reihe ist und keine möglichen Züge mehr hat, muss er so lange seinen Zug passen, bis er wieder einen legalen Zug hat. Das Spiel wird so lange fortgesetzt, bis es keine möglichen Züge mehr für die beiden Spieler gibt [10], [12]. Am Ende gewinnt der Spieler, der mehr Steine auf dem Spielbrett hat. Wenn beide Spieler die gleiche Anzahl von Steinen haben, endet das Spiel unentschieden.

3.2. Terminologie

Ecken und Kanten des Spielbretts haben eine entscheidende Bedeutung für den Erfolg in einem Othello-Spiel. Allgemein gelten die Ecken, in Abb. 3.2 die Felder $a1$, $h1$, $a8$ und $h8$, als die wertvollsten Felder im Spiel. Sie sind am schwierigsten zu übernehmen und können nach einer Übernahme nicht mehr umgedreht werden. Die Kanten haben auch eine strategische Bedeutung für die Ecken. Durch Kanten hat man eine höhere Mobilität im Spiel, siehe Kapitel 4.4. Damit kann der Spieler, der die Kanten kontrolliert, auch das Spiel besser kontrollieren. Durch diese Kontrolle ist man in der Lage Ecken und anschließend das Spiel zu gewinnen.

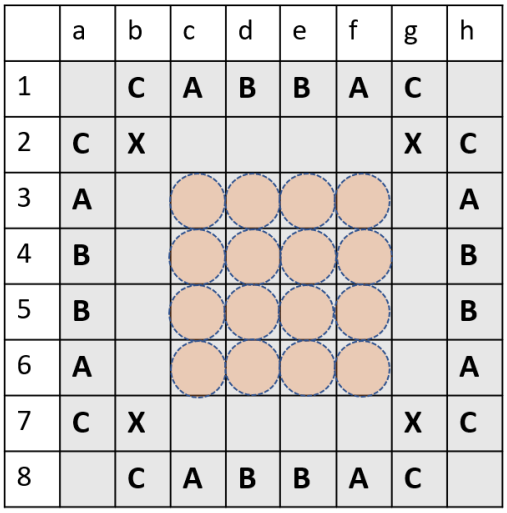


Abbildung 3.2.: Terminologie im Spiel, Bild erzeugt mit Vorlage aus [12]

Die auf Kanten liegende Felder, werden nach ihrer Nähe an Ecken als *C*-, *A*- bzw. *B*-Felder genannt [2]. Es gibt auch die sogenannten *X*-Felder, die zwar nicht auf Kanten liegen, aber die Ecken diagonal angrenzen [10]. *C*- und *X*-Felder können zum sofortigen Verlust der angrenzenden Ecke führen. In der Regel vermeiden die Spieler auf solche Felder aufgrund dieser Tatsache zu spielen, es sei denn, es gibt keine andere Möglichkeit mehr. Die orange markierten sechzehn Felder in Abb. 3.2, die sich in der Mitte des Spielbretts befinden, werden als *Sweet Sixteen* bezeichnet [12]. Im Vergleich zu den Ecken und Kanten gelten sie aber als weniger bedeutend. Gemäß den Spielregeln werden diese Felder in der Regel zu Beginn des Spiels besetzt.

4. Evaluationsmethoden

Spiele sind in der Regel sehr komplex und sie haben meistens so viele Zustände. Aufgrund der Einschränkungen von Computern, wie z.B. des Speichers, können Suchalgorithmen, siehe Kapitel 5, jedoch nicht alle Zustände eines Spiels analysieren. Algorithmen müssen also einige Züge im Spiel ignorieren oder sich auf die besten Züge des Spiels konzentrieren. Dafür benötigen die Suchalgorithmen Evaluationsmethoden, um den aktuellen Stand des Spiels zu beurteilen. Eine Evaluationsmethode kann verschiedene Aspekte eines Spiels bewerten bzw. analysieren. Sie könnte z.B. versuchen die Anzahl der Steine, die ein Spieler auf dem Brett hat, zu maximieren, siehe Kapitel 4.3, oder auch versuchen, die Position der Steine auf dem Brett zu bewerten, siehe Kapitel 4.1, wobei bestimmte Positionen wertvoller sind als andere [2], [5], [9], [15].

In dieser Arbeit werden verschiedene Evaluationsmethoden und Suchalgorithmen miteinander verglichen, indem sie in einem selbst implementierten Othello-Spiel angewendet werden. Für den Vergleich von Evaluationsmethoden wurden in dieser vorliegenden Arbeit mehrere wissenschaftliche Arbeiten herangezogen, in denen diese Methoden ausführlich beschrieben wurden [9], [15]. Es ist zu beachten, dass die in dieser Arbeit verwendeten Evaluationsmethoden größtenteils den genannten wissenschaftlichen Arbeiten entnommen und implementiert wurden.

Im Folgenden werden einige dieser Evaluationsmethoden und ihre Funktionsweise anhand von Pseudocodes kurz erläutert, die in dieser vorliegenden Arbeit implementiert und dann anschließend miteinander verglichen wurden. Für die Ergebnisse siehe Kapitel 7.2.

4.1. Statische Gewichte

Bei dieser Evaluationsmethode wird das Spielbrett in verschiedene Bereiche unterteilt, denen jeweils bestimmte Gewichte zugewiesen werden [2], [6], [15]. Diese Gewichte basieren auf der Erfahrung menschlicher Spieler und sind so gewählt, dass sie gute Spielzüge fördern und schlechte vermeiden, siehe Kapitel 3.2. Ein Beispiel für solche Gewichte in einem Othello-Spiel könnte wie in Abb. 4.1 dargestellt aussehen [2].

Die Evaluationsmethode bewertet den aktuellen Stand des Spielbretts gemäß Alg. 1. Dabei werden für den schwarzen und weißen Spieler jeweils Punkte berechnet. Dazu wird das Spielbrett in einer Schleife durchgelaufen. Wenn ein Spieler seinen Stein an einer bestimmten Position hat, wird der Wert dieser Position aus der vordefinierten Tabelle, siehe Abb. 4.1, entnommen und zu den Punkten des entsprechenden Spielers aufaddiert. Nachdem die Punkte für jeden Spieler berechnet wurden, wird die Differenz der Punkte ermittelt, die dann für den Vergleich der Spielzustände verwendet wird. Bei der Berechnung wird berücksichtigt, ob der aktuelle Spieler der maximierende Spieler ist. Da gemäß den Spielregeln der schwarze Spieler zuerst startet, wurde die Annahme getroffen, dass der schwarze Spieler im Spiel immer als der maximierende Spieler betrachtet wird.

	a	b	c	d	e	f	g	h
1	99	-8	8	6	6	8	-8	99
2	-8	-24	-4	-3	-3	-4	-24	-8
3	8	-4	7	4	4	7	-4	8
4	6	-3	4	0	0	4	-3	6
5	6	-3	4	0	0	4	-3	6
6	8	-4	7	4	4	7	-4	8
7	-8	-24	-4	-3	-3	-4	-24	-8
8	99	-8	8	6	6	8	-8	99

Abbildung 4.1.: Statische Gewichte innerhalb des Spielbretts, Bild erzeugt mit Vorlage aus [2]. Diese Gewichte basieren auf der Erfahrung menschlicher Spieler und sind so gewählt, dass sie gute Spielzüge fördern und schlechte vermeiden, siehe auch Kapitel 3.2. C- und X-Felder können zum sofortigen Verlust der angrenzenden Ecke führen. In der Regel vermeiden die Spieler auf solche Felder aufgrund dieser Tatsache zu spielen.

Algorithm 1 Pseudocode für die Evaluationsmethode *Statische Gewichte*, Alg. erzeugt mit Vorlage aus [15]

```
1:  function getScore(board : Board): integer;
2:      // berechne blackScore und whiteScore mit Hilfe der Tabelle.
3:      // der schwarze Spieler ist immer der maximierende Spieler.
4:      if actualPlayer == maximizingPlayer then
5:          difference = (blackScore − whiteScore);
6:      else
7:          difference = (whiteScore − blackScore);
8:      end if
9:      return difference
10: end function
11:
```

4.2. Ecken

Wie es im Kapitel 3.2 schon beschrieben wurde, sind die Ecken in einem Othello-Spiel strategisch wertvoll, weil sie nicht mehr von gegnerischen Steinen umgedreht und damit bis zum Ende des Spiels dauerhaft vom jeweiligen Spieler kontrolliert werden können. Diese Evaluationsmethode, was auch als *Corners Captured* bezeichnet wird, basiert auf dieser wichtigen Eigenschaft des Spiels und bewertet den aktuellen Stand des Spiels mit Hilfe der Anzahl der von jedem Spieler übernommenen Ecken. Dabei wird angenommen, dass ein Spieler, der mehr Ecken übernommen hat, im Vorteil gegenüber seinem Gegner ist, siehe Alg. 2. Wie aus dem Algorithmus ersichtlich ist, erhalten Spieler für jede eroberte Ecke 25 Punkte.

Algorithm 2 Pseudocode für die Evaluationsmethode *Ecken*, Alg. erzeugt mit Vorlage aus [9]

```
1:  function getScore(board : Board): integer;
2:      // berechne blackScore und whiteScore durch Zählen von Ecken pro Spieler
3:      // der schwarze Spieler ist immer der maximierende Spieler.
4:      if (blackScore + whiteScore) ≠ 0 then
5:          if actualPlayer == maximizingPlayer then
6:              return 25 * blackScore − 25 * whiteScore
7:          else
8:              return 25 * whiteScore − 25 * blackScore
9:          end if
10:     end if
11:     return 0;
12: end function
13:
```

Algorithm 3 Pseudocode für die Evaluationsmethode *Anzahl von Steinen*, Alg. erzeugt mit Vorlage aus [15]

```
1:  function getScore(board : Board): integer;
2:      // berechne blackScore und whiteScore durch Zählen von Steinen pro Spieler
3:      // der schwarze Spieler ist immer der maximierende Spieler.
4:      if actualPlayer == maximizingPlayer then
5:          return (100 * (blackScore − whiteScore))/(blackScore + whiteScore);
6:      else
7:          return (100 * (whiteScore − blackScore))/(whiteScore + blackScore);
8:      end if
9:  end function
10:
```

4.3. Anzahl von Steinen

Bei dieser Evaluationsmethode, die auch als *Coin Parity* bezeichnet wird, geht es vor allem darum, so viele eigene Spielsteine wie möglich auf dem Brett zu haben. Daher wird der aktuelle Stand des Spiels durch den Vergleich der Anzahl der Spielsteine der Spieler gemessen. Eine positive Bewertung bedeutet, dass der Spieler mehr eigene Spielsteine als der Gegner auf dem Brett hat und damit im Vorteil ist, siehe Alg. 3.

4.4. Mobilität

Bei dieser Evaluationsmethode wird versucht, herauszufinden, wie *beweglich* ein Spieler im aktuellen Spielstand ist. Ein höherer Wert wird als vorteilhafter angesehen, weil der jeweilige Spieler damit im Spiel mehr Flexibilität bzw. Möglichkeiten bekommt, um das Spiel zu kontrollieren. Um die Mobilität eines Spielers zu berechnen, wird einfach die Anzahl der legalen Züge pro Spieler gezählt. Der Grundgedanke dahinter ist, dass ein Spieler mit mehr legalen Zügen mehr Flexibilität und damit eine bessere Position im Spiel hat, siehe Alg. 4.

Algorithm 4 Pseudocode für die Evaluationsfunktion *Mobilität*, Alg. erzeugt mit Vorlage aus [15]

```
1:  function getScore(board : Board): integer;
2:      // berechne blackScore und whiteScore durch Zählen von legalen Zügen pro Spieler
3:      // der schwarze Spieler ist immer der maximierende Spieler.
4:      if (blackScore + whiteScore) ≠ 0 then
5:          if actualPlayer == maximizingPlayer then
6:              return (100 ∗ (blackScore − whiteScore))/(blackScore + whiteScore);
7:          else
8:              return (100 ∗ (whiteScore − blackScore))/(whiteScore + blackScore);
9:          end if
10:     end if
11:     return 0;
12: end function
13:
```

4.5. Statische Gewichte mit Mobilität

Diese Evaluationsmethode *StaticMobility* ist eine Kombination der in Kapitel 4.1 und 4.4 bereits beschriebenen Evaluationsmethoden. Bei dieser Methode werden die Evaluationswerte für jeden Spieler jeweils für beide Methoden einzeln berechnet und anschließend einfach aufaddiert. Diese Evaluationsmethode wurde entwickelt, um festzustellen, ob die Kombination verschiedener Evaluationsmethoden zu besseren Ergebnissen führt.

5. Suchalgorithmen

Suchalgorithmen werden in Spielen verwendet, um die besten Spielzüge zu finden. Bei komplexen Spielen ist es aber aufgrund der großen Anzahl von Zügen praktisch unmöglich alle Züge zu analysieren und den besten Zug zu finden. Mit Hilfe verschiedener Suchalgorithmen kann dieser Suchprozess effizienter durchgeführt werden. Diese Algorithmen können z.B. nur die besten Züge analysieren oder unwichtige Teile des Suchbaums ignorieren [4].

Im Folgenden werden einige dieser Suchalgorithmen und ihre Funktionsweise erläutert, die in dieser vorliegenden Arbeit implementiert und miteinander verglichen wurden. Für die Ergebnisse siehe Kapitel 7.1.

5.1. Alpha-Beta

Einer der am häufigsten verwendeten Suchalgorithmen in Spielen ist der Minimax-Algorithmus. Dabei wird von einer bestimmten Spielsituation ausgegangen und werden alle möglichen Züge analysiert. Die Minimax-Methode ist ein rekursiver Suchalgorithmus und verwendet die Depth-First-Search-Strategie [3], siehe die Abb. 5.1. Die Knoten in diesem Suchbaum repräsentieren die Spielzustände, während die Kanten zwischen ihnen die möglichen Spielzüge darstellen. Beim Minimax-Algorithmus ist es notwendig den gesamten Suchbaum durchzulaufen, um den besten Zug zu finden. Bei vielen komplexen Spielen ist das jedoch wegen der großen Anzahl möglicher Züge, der Tiefe des Suchbaums und den Grenzen von Computersystemen nicht möglich [16].

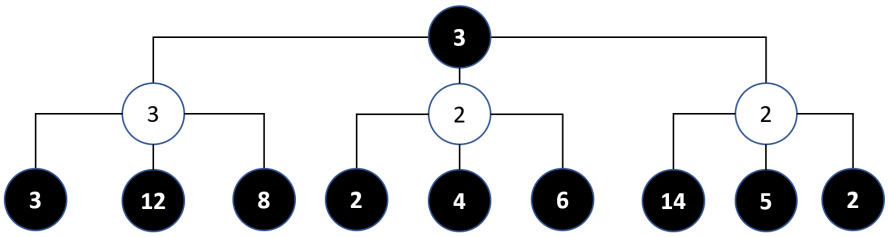


Abbildung 5.1.: Spielbaum eines Minimax-Algorithmus, Bild erzeugt mit Vorlage aus [16]. Im Bild werden MAX-Knoten in schwarzer und MIN-Knoten in weißer Farbe dargestellt. Der Minimax-Algorithmus muss den gesamten Suchbaum durchlaufen, um den besten Zug zu finden.

Der Alpha-Beta-Algorithmus (AB) kann als eine optimierte Version des Minimax-Algorithmus vorgestellt werden. Bei dieser Methode werden unwichtige Teile des Suchbaums durch eine Methode namens *Pruning* eliminiert [3], [9]. Durch das Abschneiden irrelevanter Teile des Suchbaums wird die Rechen- bzw. Laufzeit erheblich reduziert.

Der Alpha-Beta-Algorithmus verwendet α- und β-Werte, um den Suchbaum noch effizienter durchzusuchen und unnötige Teile des Baums zu verwerfen. Der α-Wert hier repräsentiert den größten bisher bekannten Wert eines MAX-Knotens, während der β-Wert den kleinsten bisher bekannten Wert eines

MIN-Knotens darstellt [1], [17]. Die α- und β-Werte werden während des gesamten Suchvorgangs aktualisiert. Es wird versucht, dass der Suchbaum so früh wie möglich abgeschnitten wird, wenn $\alpha \geq \beta$, um die Anzahl der zu bewertenden Knoten zu reduzieren, siehe Alg. 5.

Algorithm 5 Pseudocode für AB [17]

```
1:  function AB(node, depth, α, β, maxPlayer)
2:      if depth = 0 or node is a terminal node then
3:          return the heuristic value of node
4:      end if
5:      if maxPlayer then
6:          v := −INFINITY;
7:          for all child in node.children do
8:              v := max(v, AB(child, depth − 1, α, β, False))
9:              α := max(α, v)
10:             if β ≤ α then
11:                 break                          ▷ Alpha-Beta-Pruning (β cut-off)
12:             end if
13:         end for
14:     else
15:         v := +INFINITY;
16:         for all child in node.children do
17:             v := min(v, AB(child, depth − 1, α, β, True))
18:             β := min(β, v)
19:             if β ≤ α then
20:                 break                          ▷ Alpha-Beta-Pruning (α cut-off)
21:             end if
22:         end for
23:     end if
24:     return v
25: end function
```

Wenn der maximierende Spieler an der Reihe ist, wird v auf den negativen Unendlichkeitswert (-*INFINITY*) gesetzt. Dann wird eine Schleife über alle Kinderknoten des aktuellen Knotens *node* durchgeführt. Für jeden Kindknoten wird die Funktion *AB* rekursiv aufgerufen, wobei *maxPlayer* auf *False* gesetzt wird, um anzuzeigen, dass der nächste Spieler der minimierende Spieler ist. Der Wert v wird auf das Maximum von v und dem Wert, zurückgegeben von *AB* für das Kindknoten, aktualisiert. Zusätzlich wird α auf das Maximum von α und v aktualisiert. Wenn β kleiner oder gleich α ist, wird die Schleife abgebrochen, da dieser Zweig nicht weiter verfolgt werden muss (β Cutoff), siehe die Abb. 5.3.

Wenn der minimierende Spieler an der Reihe ist, wird v auf den positiven Unendlichkeitswert (+*INFINITY*) gesetzt. Ähnlich wie im vorherigen Fall wird eine Schleife über alle Kinderknoten durchgeführt. Diesmal wird v auf das Minimum von v und dem Wert, zurückgegeben von *AB* für das Kindknoten, aktualisiert. Gleichzeitig wird β auf das Minimum von β und v aktualisiert. Wenn β kleiner oder gleich α ist, wird die Schleife wieder abgebrochen (α Cutoff).

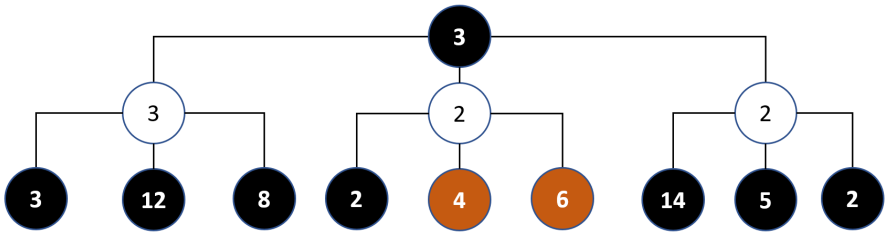


Abbildung 5.2.: *Alpha-Beta-Algorithmus mit Pruning*, Bild erzeugt mit Vorlage aus [16]. Im Bild werden MAX-Knoten in schwarzer und MIN-Knoten in weißer Farbe dargestellt. Orangene Knoten werden vom Algorithmus nicht mehr betrachtet, weil sie keine Auswirkung mehr auf das Endergebnis haben werden. Um zu sehen, wie sich die α- bzw. β-Werte ändern, siehe die schrittweise Erklärung in Abb. 5.3.

Nachdem alle Kinderknoten durchlaufen wurden, wird der Wert v zurückgegeben, der den besten Zug für den aktuellen Spieler repräsentiert.

Der Alpha-Beta-Algorithmus optimiert die Suche im Spielbaum, indem er irrelevante Zweige abschneidet (Cutoffs) und somit die Anzahl der zu untersuchenden Knoten reduziert, siehe Abb. 5.2 und Abb. 5.3. Dadurch wird die Suchzeit erheblich verkürzt. Damit wird die effiziente Berechnung des besten Zugs in Spielen mit hoher Suchtiefe ermöglicht.

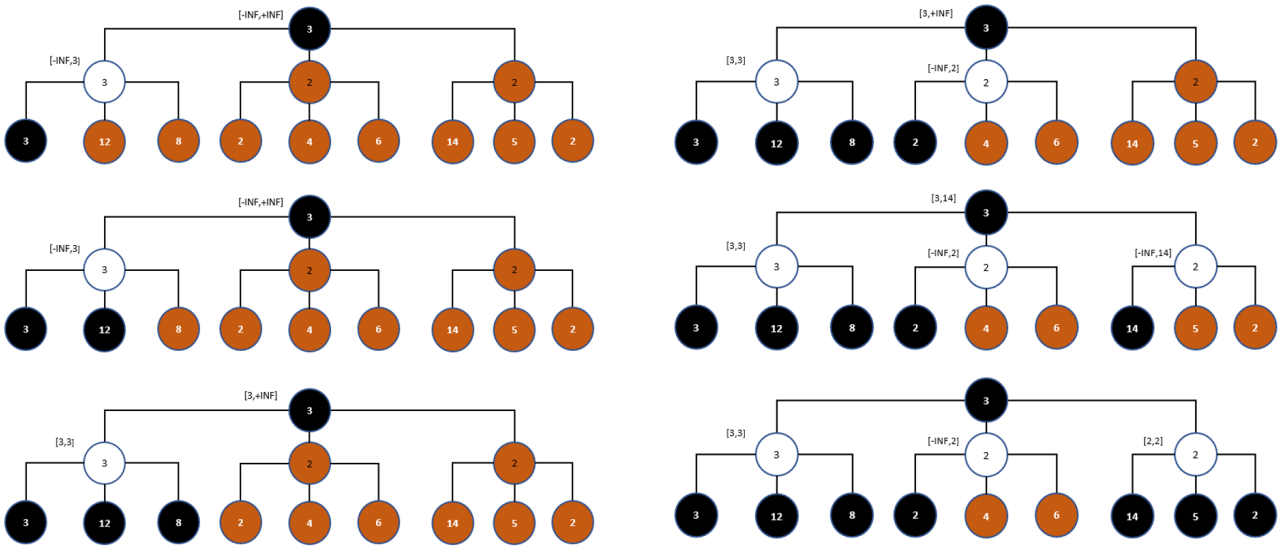


Abbildung 5.3.: *Alpha-Beta-Algorithmus mit Pruning*, Bild erzeugt mit Vorlage aus [16] und [17]. Im Bild werden MAX-Knoten in schwarzer und MIN-Knoten in weißer Farbe dargestellt. Orangene Knoten wurden entweder vom Algorithmus noch nicht betrachtet oder sie werden vom Algorithmus überhaupt nicht betrachtet, weil sie keine Auswirkung mehr auf das Endergebnis haben werden. Die Werte $[\alpha, \beta]$ am Knoten zeigen jeweils die α- bzw. β-Werte des jeweiligen Knotens.

5.2. MTD(f)

Der *MTD(f)* Algorithmus [13] ist eine effiziente Variante des Minimax-Algorithmus, die wegen seiner Einfachheit und Effizienz von vielen Spielen[1] verwendet wird. Im Kern besteht der MTD(f)-Algorithmus aus nur wenigen Zeilen Code, siehe Alg. 6.

Algorithm 6 Pseudocode für MTD(f) [13]

```
1:  function MTDF(root : node_type, f : integer, d : integer): integer;
2:      g := f;
3:      upperbound := +INFINITY;
4:      lowerbound := −INFINITY;
5:      repeat
6:          if g == lowerbound then
7:              beta := g + 1;
8:          else
9:              beta := g;
10:         end if
11:         g := AlphaBetaWithMemory(root, beta − 1, beta, d);
12:         if g < beta then
13:             upperbound := g;
14:         else
15:             lowerbound := g;
16:         end if
17:     until lowerbound ≥ upperbound;
18:     return g
19: end function
```

Die *MTDF*-Methode wird mit einem anfänglichen Schätzwert f aufgerufen, der typischerweise auf *0* gesetzt wird. Dieser Schätzwert fungiert als erste Annäherung an den optimalen Wert und wird im Verlauf der Berechnungen durch wiederholte Aufrufe der Methode *AlphaBetaWithMemory (ABWM)*, siehe Alg. 7, verfeinert. Diese iterative Verfeinerung des Schätzwerts ermöglicht eine schnellere Annäherung an den tatsächlichen besten Wert. Ein sorgfältig gewählter Startwert f beschleunigt die Konvergenz des Algorithmus erheblich, da er eine effizientere Anpassung des Fensters [LOWERBOUND, UPPERBOUND] in der *MTDF*-Methode ermöglicht und somit die Anzahl der benötigten Schleifendurchläufe verringert, wie in Abb. 5.4 veranschaulicht. Es ist jedoch zu beachten, dass eine ungenaue Wahl des Startwerts dazu führen kann, dass der Algorithmus mehr Schleifendurchläufe benötigt, um den tatsächlichen besten Wert zu ermitteln.

Beim Aufruf der *ABWM*-Methode ist der α-Wert immer um eins kleiner als der β-Wert, weshalb diese Methode als *Zero Window* bezeichnet wird. In seiner Arbeit beschrieb *Plaat* [13], dass solche schmalen Fenster zu mehr *Cutoffs* führen und dadurch die Suche effizienter gestalten.

Jeder Aufruf der Methode ABWM liefert einen neuen Grenzwert, der das Intervall [LOWERBOUND, UPPERBOUND] in der MTDF-Methode weiter einschränkt, in der der beste Wert gesucht wird, siehe

[1] z.B. Cilkchess: Ein Schachprogramm, was an der MIT entwickelt wurde, https://www.chessprogramming.org/CilkChess, (abgerufen am 17.08.2023)

Algorithm 7 Pseudocode für AlphaBetaWithMemory [13]

```
1:  function AlphaBetaWithMemory(n : node_type, alpha, beta, d : integer): integer;
2:      if retrieve(n) == OK then                          ▷ Transposition table lookup
3:          if n.lowerbound ≥ beta then
4:              return n.lowerbound;
5:          end if
6:          if n.upperbound ≤ alpha then
7:              return n.upperbound;
8:          end if
9:          alpha := max(alpha, n.lowerbound);
10:         beta := min(beta, n.upperbound);
11:     end if
12:     if d == 0 then
13:         g := evaluate(n);                                  ▷ leaf node
14:     else if n == MAXNODE then
15:         g := −INFINITY;
16:         a := alpha;                                ▷ save original alpha value
17:         c := firstchild(n);
18:         while (g < beta) and (c ≠ NOCHILD) do
19:             g := max(g, AlphaBetaWithMemory(c, a, beta, d − 1));
20:             a := max(a, g);
21:             c := nextbrother(c);
22:         end while
23:     else                                               ▷ n is a MINNODE
24:         g := +INFINITY;
25:         b := beta;                                 ▷ save original beta value
26:         c := firstchild(n);
27:         while (g > alpha) and (c ≠ NOCHILD) do
28:             g := min(g, AlphaBetaWithMemory(c, alpha, b, d − 1));
29:             b := min(b, g);
30:             c := nextbrother(c);
31:         end while
32:     end if
33:                                  ▷ Traditional transposition table storing of bounds
34:     if g ≤ alpha then                    ▷ Fail low result implies an upper bound
35:         n.upperbound := g;
36:         store n.upperbound;
37:     end if
38:                 ▷ Found an accurate minimax value – will not occur if called with zero window
39:     if g > alpha and g < beta then
40:         n.lowerbound := g;
41:         n.upperbound := g;
42:         store n.lowerbound, n.upperbound;
43:     end if
44:     if g ≥ beta then                     ▷ Fail high result implies a lower bound
45:         n.lowerbound := g;
46:         store n.lowerbound;
47:     end if
48:     return g;
49: end function
```

Startwerte: f = g = beta = 0		
beta	Grenzwert von ABWM = g	Aktion
0	-1	[g < beta]: UPPERBOUND: 2147483647 -> -1
-1	-2	[g < beta]: UPPERBOUND: -1 -> -2
-2	-3	[g < beta]: UPPERBOUND: -2 -> -3
-3	-4	[g < beta]: UPPERBOUND: -3 -> -4
-4	-5	[g < beta]: UPPERBOUND: -4 -> -5
-5	-6	[g < beta]: UPPERBOUND: -5 -> -6
-6	-7	[g < beta]: UPPERBOUND: -6 -> -7
-7	-7	[g >= beta]: LOWERBOUND: -2147483648 -> -7
Ergebnis = -7		

Startwerte: f = g = beta = 5		
beta	Grenzwert von ABWM = g	Aktion
5	4	[g < beta]: UPPERBOUND: 2147483647 -> 4
4	3	[g < beta]: UPPERBOUND: 4 -> 3
3	2	[g < beta]: UPPERBOUND: 3 -> 2
2	1	[g < beta]: UPPERBOUND: 2 -> 1
1	0	[g < beta]: UPPERBOUND: 1 -> 0
0	-1	[g < beta]: UPPERBOUND: 0 -> -1
-1	-2	[g < beta]: UPPERBOUND: -1 -> -2
-2	-3	[g < beta]: UPPERBOUND: -2 -> -3
-3	-4	[g < beta]: UPPERBOUND: -3 -> -4
-4	-5	[g < beta]: UPPERBOUND: -4 -> -5
-5	-6	[g < beta]: UPPERBOUND: -5 -> -6
-6	-7	[g < beta]: UPPERBOUND: -6 -> -7
-7	-7	[g >= beta]: LOWERBOUND: -2147483648 -> -7
Ergebnis = -7		

Startwerte: f = g = beta = -5		
beta	Grenzwert von ABWM = g	Aktion
-5	-6	[g < beta]: UPPERBOUND: 2147483647 -> -6
-6	-7	[g < beta]: UPPERBOUND: -6 -> -7
-7	-7	[g >= beta]: LOWERBOUND: -2147483648 -> -7
Ergebnis = -7		

Startwerte: f = g = beta = -7		
beta	Grenzwert von ABWM = g	Aktion
-7	-7	[g >= beta]: LOWERBOUND: -2147483648 -> -7
-6	-7	[g < beta]: UPPERBOUND: 2147483647 -> -7
Ergebnis = -7		

Abbildung 5.4.: Mit einem gut ausgewählten Startwert f kann der Algorithmus schneller an den besten Wert konvergieren. Damit kann das Fenster [LOWERBOUND, UPPERBOUND] in der *MTDF*-Methode effizienter angepasst und die Anzahl der erforderlichen Durchläufe reduziert werden. Eine ungenaue Wahl des Startwerts kann jedoch dazu führen, dass der Algorithmus mehr Durchläufe benötigt, um den tatsächlichen besten Wert zu finden. In der Abb. ist es zu sehen, wie schnell man mit 4 unterschiedlichen Startwerten für f zum Ergebnis kommt, hier $0, 5, -5$ und -7. Wählt man als Startwert den richtigen Wert aus, hier -7, ist maximal 2 Durchläufe ausreichend.

Abb. 5.4. Dieser iterative Ansatz ermöglicht es, sich mit jedem Aufruf dem tatsächlichen besten Wert schrittweise zu nähern. Allerdings gibt es einen Nachteil. Durch wiederholte Aufrufe der ABWM-Methode werden bestimmte Teile des Suchbaums mehrmals durchsucht, was zu unnötigem Rechenaufwand führt.

Um dieses Problem zu umgehen, wird im Algorithmus die Verwendung einer *Transpositionstabelle*[2] vorgeschlagen. Diese Tabelle dient dazu, Spielzustände zusammen mit den gefundenen Grenzwerten zu speichern. Dadurch wird verhindert, dass dieselben Suchen wiederholt durchgeführt werden müssen. Die Herausforderung besteht jedoch darin, den Spielzustand in der Tabelle effizient zu speichern bzw. abzurufen.

In dieser vorliegenden Arbeit wurde die *Zobrist-Hashing*-Methode zur Speicherung von Spielständen in Transpositionstabellen ausgewählt, wie im Kapitel 5.2.1 näher erläutert wird. Durch die Verwendung von Zobrist-Hashing können Spielstellungen effizient in einer Transpositionstabelle gespeichert werden. Dabei wird jeder Spielstellung ein eindeutiger Hash-Wert zugewiesen, der dazu dient, bereits gespeicherte Spielstellungen schnell zu identifizieren. Die Funktionsweise des MTD(f)-Algorithmus in Verbindung mit Zobrist-Hashing wird in Abb. 5.5 grob skizziert.

Durch die Nutzung von ABWM in Verbindung mit Transpositionstabellen und Zobrist-Hashing kann der Algorithmus effizienter und schneller den besten Zug finden und dabei bereits besuchte Spielzustände wiederverwenden. Dies trägt zur Steigerung der Leistungsfähigkeit des MTD(f)-Algorithmus erheblich bei.

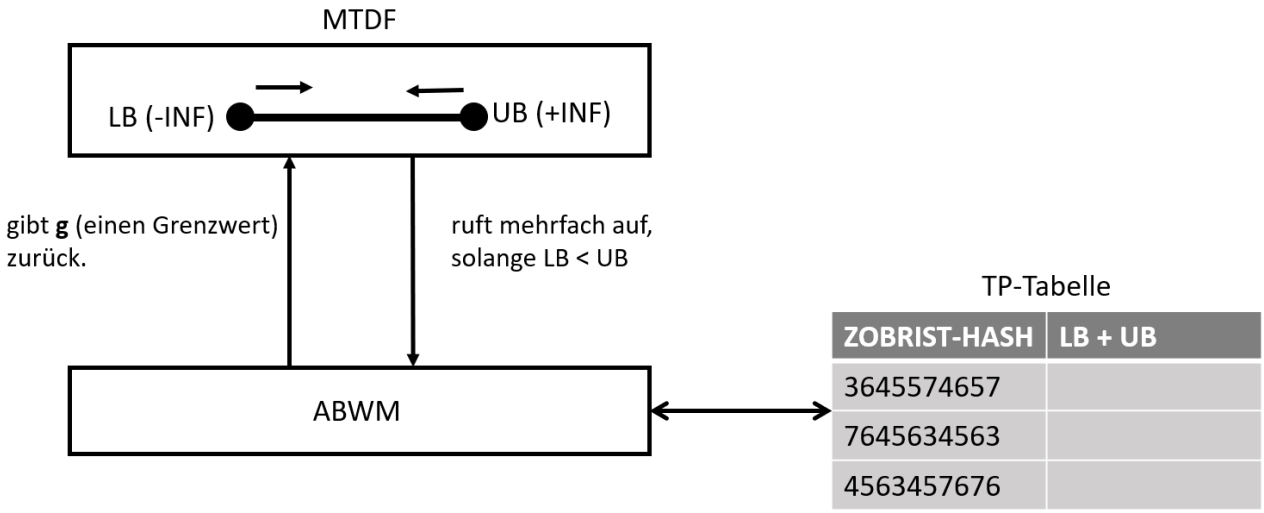


Abbildung 5.5.: Funktionsweise des *MTD(f)*-Algorithmus. Durch das Einsetzen einer Transpositionstabelle wird verhindert, dass die gleichen Spielzustände mehrfach gesucht werden. Um den Stand des Spiels in der Transpositionstabelle zu speichern braucht man eine Hashing-Methode. In der vorliegenden Arbeit wurde diesbezüglich für die *Zobrist-Hashing*-Methode entschieden, siehe Kapitel 5.2.1

Wenn ein Grenzwert, der durch die *ABWM*-Methode gefunden wird, niedriger ausfällt ($g \leq alpha$, siehe Alg. 7), wird das als *fail low* bezeichnet. In solchen Fällen deutet dies darauf hin, dass die

[2]Zugfolgen, die zur gleichen Stellung führen, werden als Transpositionen bezeichnet [5], [14].

MTDF-Methode die obere Grenze *UPPERBOUND*, bezeichnet als f^+, auf den durch die *ABWM*-Methode gefundenen Grenzwert g reduzieren soll. Wenn der Grenzwert im Gegensatz dazu jedoch hoch ausfällt ($g \geq beta$, siehe Alg. 7), wird dies als *fail high* bezeichnet. In diesem Fall sollte die *MTDF*-Methode die untere Grenze *LOWERBOUND*, bezeichnet als f^-, auf den durch die *ABWM*-Methode gefundenen Grenzwert g erhöhen. Damit wird das Intervall [LOWERBOUND, UPPERBOUND] immer kleiner. Wenn sowohl die obere als auch die untere Grenze aufeinandertreffen, wird der Minimax-Wert ermittelt, wie in Abb 5.4 dargestellt.

Die von der ABWM-Methode ermittelten Grenzwerte werden vor ihrer Übertragung an die MTDF-Methode in der Transpositionstabelle abgelegt. Bei einem *fail low* wird der ermittelte Grenzwert als *UPPERBOUND* gespeichert, während im Fall eines *fail high* er als *LOWERBOUND* gespeichert wird. Diese Einträge in der Transpositionstabelle dienen dazu, zu verhindern, dass identische Suchvorgänge später erneut durchgeführt werden müssen, siehe Alg. 7.

Im Listing 5.1 wird es grob demonstriert, wie die Transpositionstabelle in der ABWM-Methode implementiert ist. Hier ist es zu erkennen, dass die *Zobrist-Keys* bereits bei der Objekterstellung vor dem Start des Spiels initialisiert werden. Eine ausführlichere Erläuterung zu den *Zobrist-Keys* findet sich im Kapitel 5.2.1. Um in der Transpositionstabelle nicht nur den Hash-Wert, sondern auch die dazugehörigen Unter- und Obergrenzen (also LB und UB) zu speichern, wurde der komplexe Datentyp *TranspositionTableEntry* definiert. Dieser Datentyp erleichtert das Speichern der Grenzwerte erheblich, indem er die Methode *addToTranspositionTable* verwendet, um sie in die Tabelle einzutragen. Besonders hervorzuheben ist, dass neben den Grenzwerten auch die zugehörigen Suchtiefen in der Transpositionstabelle festgehalten werden. Diese zusätzlichen Informationen erleichtern das spätere Abrufen der Grenzwerte aus der Transpositionstabelle erheblich. Denn nur diejenigen Einträge in der Tabelle müssen ausgewertet werden, die entweder für die exakt gleiche Suchtiefe oder für größere Suchtiefen gespeichert wurden. In diesem Kontext bedeutet eine größere Suchtiefe, dass die betrachteten Knoten näher an den Blattknoten liegen und somit eine bessere Perspektive auf das Spiel bieten.

Bei jedem Aufruf der ABWM-Methode wird zunächst der Hash-Wert des aktuellen Spielstands, repräsentiert durch das *board*-Objekt, berechnet. Sollte dieser Spielstand bereits in der Transpositionstabelle gespeichert sein, wird der zuvor ermittelte Grenzwert, entweder *LB* für *fail high* oder *UB* für *fail low* zurückgegeben. Die Entscheidung darüber, welcher Grenzwert zurückgegeben wird, basiert auf den bereits zuvor erklärten Kriterien von *fail low* und *fail high*, wie sie auch beim Speichern in die Transpositionstabelle verwendet wurden, siehe Listing 5.1. Konkret bedeutet dies, dass ein *UB* zurückgegeben wird, wenn der gefundene Grenzwert niedriger ausfällt, also ein *fail low* vorliegt, während ein *LB* zurückgegeben wird, wenn der gefundene Grenzwert höher ausfällt, also ein *fail high* auftritt.

```
1 public class MTDF implements Algorithm {
2
3     // andere Codes hier
4
5     ZobristHash zobristHash = new ZobristHash(Constants.GRID_SIZE);
6     private Map<Integer, TranspositionTableEntry> transpositionTable =
          new HashMap<>();
7
8     // Methode, die einen neuen Eintrag in die Transposotionstabelle
```

```
        schreibt
9       private void addToTranspositionTable(int hash, int lowerbound, int
            upperbound, int depth) {
10          transpositionTable.put(hash, new TranspositionTableEntry(
                lowerbound, upperbound, depth));
11      }
12
13      public Position getMove(Board board, Player player, Evaluation
            blackEvaluation, Evaluation whiteEvaluation) {
14
15          // andere Codes hier
16
17          // Initial-Wert für MTDF
18          int f = 0;
19          int bestValue = mtdf(board, f, Constants.DEPTH);
20
21          // andere Codes hier
22      }
23
24      private int mtdf(Board board, int f, int depth) {
25          // mehrfach AlphaBetaWithMemory aufrufen und g herausfinden
26          return g;
27      }
28
29      private int AlphaBetaWithMemory(Board board, int alpha, int beta, int
            depth) {
30
31          // den Hash-Wert für das board-Objekt berechnet
32          int hash = zobristHash.getHash(board);
33
34          // Transpositionstabelle (TT) nachschlagen. Hat die TT den Hash-
                Wert?
35          TranspositionTableEntry tableEntry = transpositionTable.get(hash)
                ;
36          if (tableEntry != null) {
37
38              // untere oder obere Grenze zurückgeben
39
40              // Ein niedriges Ergebnis(fail low) impliziert eine obere
                    Grenze. Aus der Transpositiontabelle werden nur die Einträ
                    ge aus der selben Tiefe oder aus größeren Tiefen
                    ausgelesen. Tiefe größer bedeutet, dass die Knoten näher
                    an Blatt-Knoten sind.
41              if (tableEntry.getUpperbound() <= alpha && (Constants.DEPTH -
                     depth) <= tableEntry.getDepth()) {
42                  return tableEntry.getUpperbound();
43              }
44
45              // Ein höheres Ergebnis(fail high) impliziert eine untere
                    Grenze. Aus der Transpositiontabelle werden nur die Einträ
```

```
                ge aus der selben Tiefe oder aus größeren Tiefen
                ausgelesen. Tiefe größer bedeutet, dass die Knoten näher
                an Blatt-Knoten sind.
46              if (tableEntry.getLowerbound() >= beta && (Constants.DEPTH -
                depth) <= tableEntry.getDepth()) {
47                  return tableEntry.getLowerbound();
48              }
49          }
50
51          // Code für die Alpha-Beta-Suche hier
52
53          // Ein niedriges Ergebnis (fail low) impliziert eine obere Grenze
54          if (g <= alpha) {
55              addToTranspositionTable(hash, Integer.MIN_VALUE, g,
                    Constants.DEPTH - depth);
56          }
57          // Ein höheres Ergebnis (fail high) impliziert eine untere Grenze
58          if (g >= beta) {
59              addToTranspositionTable(hash, g, Integer.MAX_VALUE, Constants
                    .DEPTH - depth);
60          }
61
62          return g;
63      }
64  }
```

Listing 5.1: Implementierung der Transpositionstabelle

Im Folgenden wird noch auf die Zobrist-Hashing-Methode eingegangen, welche in den Transpositionstabellen angewendet wird. Diese Methode ist von entscheidender Bedeutung, da sie dazu dient, die effiziente Speicherung und schnelle Abfrage von Spielstellungen in den Transpositionstabellen zu ermöglichen.

5.2.1. Zobrist Hashing

Zobrist-Hashing [4], [5], [18] ist eine bekannte Methode zur Speicherung von Spielstellungen. Bei dieser Methode wird jeder Spielfeldposition vor Spielbeginn eine zufällig generierte Zahl zugewiesen, genannt *Zobrist-Keys*, und zwar für jede mögliche Spielfigur. Ein Beispiel dafür ist in Listing 5.2 dargestellt. Im Fall von Othello, einem Spiel mit einem 8x8 Spielbrett und den beiden möglichen Spielfiguren Schwarz und Weiß, genügt ein Array der Größe [8][8][2], um die erforderlichen Zobrist-Keys für das Othello-Spiel zu speichern.

```
1  public class ZobristHashing {
2
3      private final int[][][] zobristKeys;
4
5      // andere Codes hier
6
7      // Initialisiere die Zobrist-Keys mit zufälligen Werten
```

```
    public initZobristKeys() {
        zobristKeys = new int[dimension][dimension][ANZAHL_FIGUR];
        for (int i = 0; i < dimension; i++) {
            for (int j = 0; j < dimension; j++) {
                for (int k = 0; k < ANZAHL_FIGUR; k++) {
                    zobristKeys[i][j][k] = <ZUFAELLIGE_ZAHL>
                }
            }
        }
    }

    // andere Codes hier
}
```

Listing 5.2: Initialisierung von Zobrist-Keys mit zufälligen Werten vor Spielbeginn.

Der *Hash*-Wert eines Spielstands wird durch die Anwendung der *XOR*-Operation (binäres Exklusiv-Oder) auf die Zobrist-Keys aller Spielfelder berechnet, wie im Beispiel-Code in Listing 5.3 dargestellt. Es ist wichtig zu beachten, dass bei dieser Berechnung leere Spielfelder ignoriert werden. Der ermittelte Hash-Wert wird anschließend in der entsprechenden Transpositionstabelle gespeichert, wie in Abbildung 5.5 veranschaulicht. Dieser Prozess gewährleistet, dass jedem Spielstand ein eindeutiger Hash-Wert zugeordnet werden kann.

```

public class ZobristHashing {

    // Zobrist-Keys im Vorfeld berechnen
    private final int[][][] zobristKeys;

    // andere Codes hier

    public int getHash(Board board) {
        int hash = 0;

        for (int i = 0; i < dimension; i++) {
            for (int j = 0; j < dimension; j++) {

                Cell cell = board.getCell(i, j);

                // die leeren Felder ignorieren
                if (cell == Cell.BLACK) {
                    hash = hash ^ zobristKeys[i][j][BLACK];
                } else if (cell == Cell.WHITE) {
                    hash = hash ^ zobristKeys[i][j][WHITE];
                }
            }
        }
        return hash;
    }

```

```
28      // andere Codes hier
29  }
```

Listing 5.3: Berechnung des Hash-Wertes eines Spielstands durch die Kombination der Zobrist-Keys aller Spielfelder mittels der *XOR*-Operation (binäres Exklusiv-Oder). Es ist wichtig zu beachten, dass bei dieser Berechnung leere Spielfelder ignoriert werden.

Eine wichtige Eigenschaft des Zobrist-Hashings besteht darin, dass es die Berechnung eines neuen Hash-Werts aus einem vorhandenen Hash-Wert ermöglicht [5]. Diese Eigenschaft vereinfacht die Aktualisierung des Hash-Werts nach einem Zug, ohne dass eine erneute Durchquerung des gesamten Spielbretts erforderlich ist. Dies beruht auf der Umkehrbarkeit der XOR-Operation. Bei der Berechnung des Hash-Werts des aktuellen Spielstands werden die Werte jeder besetzten Zelle auf dem Spielbrett, die durch *Zobrist-Keys* repräsentiert werden, mittels XOR-Operation in die Berechnung einbezogen. Durch eine weitere XOR-Operation kann der Beitrag einer einzelnen Zelle ohne Auswirkungen auf die anderen Zellen rückgängig gemacht werden. Dies ermöglicht eine einfache Ableitung neuer Hash-Werte nach einem Zug aus den vorherigen Hash-Werten, wie im Beispiel in Listing 5.4 verdeutlicht.

```
1
2       @Test
3       void getZobristHashTest() {
4
5           Integer hash1 = o.getHash(board);
6
7           // gibt -1075959010 aus
8           System.out.println("Hash1: " + hash1);
9
10          // andere Codes hier. Board entsprechend ändern.
11
12          int calculatedHash = hash1 ^ zobristKeys[2][3][BLACK];
13          calculatedHash = calculatedHash ^ zobristKeys[3][3][WHITE];
14          calculatedHash = calculatedHash ^ zobristKeys[3][3][BLACK];
15
16          // gibt 1117115496 aus.
17          System.out.println("Berechnung über Hash1: " + calculatedHash);
18
19          Integer hash2 = o.getHash(board);
20
21          // gibt 1117115496 aus.
22          System.out.println("Hash2: " + hash2);
23      }
```

Listing 5.4: Berechnung des neuen Hash-Werts durch den alten Hash-Wert

In diesem Beispiel werden die Spielsituationen in den Abbildungen 3.1.*a* und 3.1.*b* betrachtet. In diesem Szenario war der schwarze Spieler am Zug. Angenommen, der schwarze Spieler zieht nach [3,d]. Dies würde die Situation in Abbildung 3.1.*b* erzeugen. Um von Zustand *a* zu Zustand *b* zu gelangen, muss die schwarze Spielfigur auf das zuvor leere Feld [3,d] platziert werden. Da sich in Zelle [4,d] eine weiße Spielfigur befindet, muss die weiße Figur zunächst aus dieser Zelle entfernt werden. Erst danach kann die schwarze Figur in das Feld [4,d] gesetzt werden. In dieser Spielsituation können wir

den Hash-Wert vor dem Zug verwenden, um den Hash-Wert nach dem Zug zu berechnen, ohne das gesamte Spielbrett erneut durchlaufen zu müssen, ähnlich wie im Beispiel in Listing 5.4. In diesem Fall ist zu erkennen, dass der berechnete Hash-Wert über *hash1* und *hash2* identisch sind.

Obwohl das Zobrist-Hashing eine einfache Methode zur Ableitung des Hash-Werts des aktuellen Spielstands aus dem vorherigen Spielstand bietet, wurde diese Eigenschaft in der vorliegenden Arbeit bewusst nicht genutzt. Dies resultiert aus der Tatsache, dass im Othello-Spiel eine große Anzahl von Steinen in einem einzigen Zug an den gegnerischen Spieler übergehen kann, was im Vergleich zu anderen Spielen wie Schach die Anwendung dieser Methode erschwert. Dies führt dazu, dass der Hash-Wert des aktuellen Spielstands jedes Mal neu berechnet werden muss, indem das gesamte Spielbrett durchlaufen wird, was sich negativ auf die benötigte Rechenzeit auswirkt.

5.3. Best-First-Minimax

Der Best-First-Minimax-Algorithmus (BFM) [8] ist ebenfalls eine optimierte Version des Minimax-Algorithmus, siehe Alg. 8. Im Gegensatz zu Minimax-Algorithmus, wo alle möglichen Züge analysiert werden, werden hier nur die besten Züge analysiert. Um diese zu finden, müssen zunächst die aktuell möglichen Züge mit Hilfe einer Evaluationsmethode bewertet werden.

Die Grundidee hinter diesem Algorithmus ist es, die besten Züge zuerst auszuwerten. Algorithmus wählt dann denjenigen Zug aus, von dem er denkt, dass er am besten ist. Dabei merkt er sich den besten Zug, den er gemacht hat, und versucht diesen Weg, bekannt auch als *Principal Variation (PV)*, weiterzugehen, um noch bessere Ergebnisse zu erzielen. Der Blattknoten, der entlang der PV als nächstes expandiert werden soll, wird als *Principal Leaf (PL)* bezeichnet.

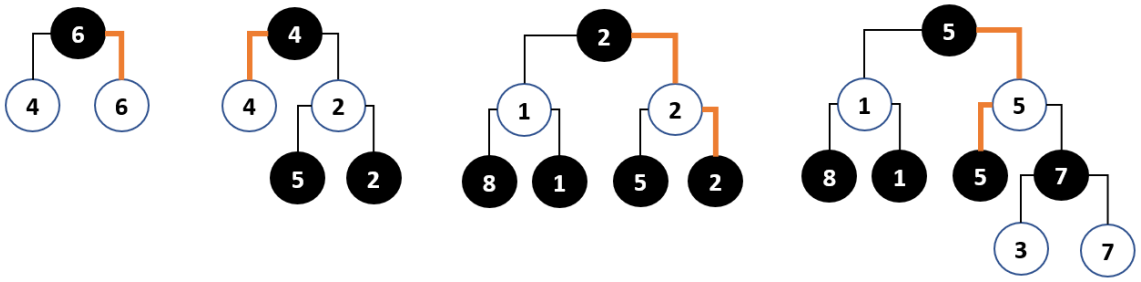


Abbildung 5.6.: *Best-First-Minimax-Algorithmus (BFM)*, Bild erzeugt mit Vorlage aus [8]. Im Bild werden MAX-Knoten in schwarzer und MIN-Knoten in weißer Farbe dargestellt. Die orangen Linien repräsentieren die *principal variation (PV)*. Der Nachteil dieser Vorgehensweise ist der exponentiell steigende Speicherbedarf in Abhängigkeit von der Baumtiefe.

Wenn ein Knoten expandiert wird, werden zuerst alle seine Kinder durch eine Evaluationsmethode ausgewertet. Durch die Expansion ändert sich auch der Wert des expandierten Knotens. Abhängig vom Typ des expandierten Knotens, ob MAX oder MIN, erhält der Knoten entweder den Wert des maximalen oder minimalen Kindknotens. Der Algorithmus setzt dann seine Bewegung im Baum nach oben fort und aktualisiert die Werte seiner Vorfahren, bis er entweder die Wurzel oder einen Knoten erreicht, dessen Wert sich nicht ändert. Durch die aktualisierten Knotenwerte ändern sich auch PV und PL. Dadurch navigiert der Algorithmus den Baum erneut hinunter zu einem maximalwertigen

Kindknoten eines MAX-Knotens oder einem minimalwertigen Kindknoten eines MIN-Knotens, bis er einen neuen Blattknoten bzw. PL erreicht [8]. Dieser wird als nächstes erweitert bzw. expandiert, siehe Abbildung 5.6.

Eine Herausforderung dieser Methode ist jedoch der exponentiell steigende Speicherbedarf in Abhängigkeit von der Tiefe des Baums. Um dieses Problem zu bewältigen, haben die Entwickler eine verbesserte Variante des Algorithmus namens *Recursive Best-First Minimax Search (RBFMS)* vorgeschlagen, welche auch in Abb. 5.7 veranschaulicht wird, siehe Alg. 8. In der vorliegenden Arbeit wurde diese Variante des Algorithmus bevorzugt und implementiert.

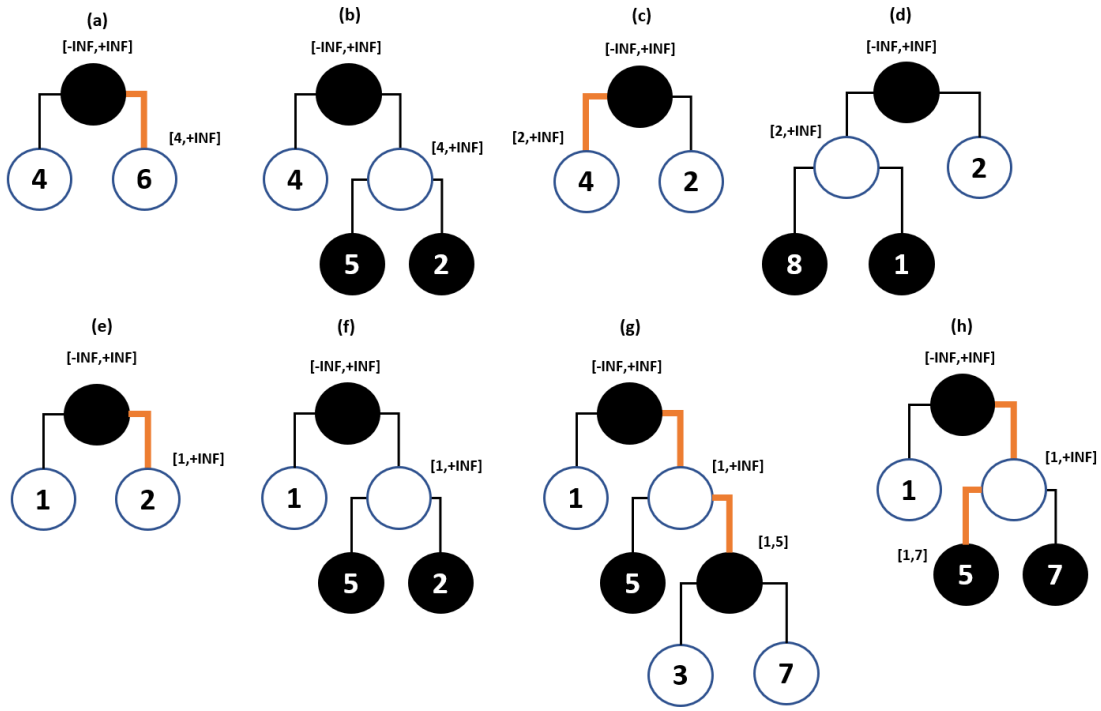


Abbildung 5.7.: *Recursive-Best-First-Minimax*, Bild erzeugt mit Vorlage aus [8]. Werte an Knoten, also [a,b], stehen jeweils für α- und β-Werte des jeweiligen Knotens. Im Bild werden MAX-Knoten in schwarzer und MIN-Knoten in weißer Farbe dargestellt. Die orangen Linien repräsentieren die *principal variation (PV)*.

Ähnlich dem Minimax-Algorithmus besteht der Algorithmus aus zwei Methoden: *BFMAX* und *BFMIN*. Diese rufen sich gegenseitig rekursiv auf. Obwohl es im ursprünglichen Pseudocode nicht explizit angegeben ist, endet die Rekursion durch das Erreichen einer bestimmten Tiefe, ähnlich wie bei anderen Algorithmen. Zu Beginn verwendet der Algorithmus wie der Alpha-Beta-Algorithmus $-\infty$ als α und $+\infty$ als β.

Bei der Expansion eines Knotens werden zunächst seine Kindknoten mithilfe einer Evaluationsmethode bewertet. Während dieser Bewertung werden die Werte direkt mit α bzw. β verglichen. Wenn z.B. der Evaluationswert bei einem MAX-Knoten größer als β oder bei einem MIN-Knoten kleiner als α ist, wird dieser Wert direkt zurückgegeben. Dies entspricht den *Cutoffs*, die man vom Alpha-Beta-Algorithmus kennt. Damit werden unnötige Zweige frühzeitig abgeschnitten. Wenn jedoch alle Evaluationswerte zwischen α und β liegen, wird der beste Zug ausgewählt, der als Nächstes expandiert werden soll. Dafür müssen alle möglichen Züge untereinander sortiert werden. Bei BFMAX

werden alle Züge absteigend und bei BFMIN aufsteigend sortiert.

Im Algorithmus sieht man in beiden Methoden, dass die Rekursion solange durchgeführt wird, wie der Wert des ersten Kindknotens zwischen den Grenzen α und β liegt. Mit jeder Rekursion wird der α-Wert größer und der β-Wert kleiner. Dies führt dazu, dass das Suchfenster immer weiter eingeschränkt wird, innerhalb dessen gesucht wird. Es ist auch zu beachten, dass bei der gegenseitigen Rekursion von BFMAX und BFMIN auch der Wert des zweitbesten Knotens, im Pseudocode *M[2]*, verwendet wird. Dieser wird beim Aufruf der BFMIN-Methode als Untergrenze bzw. α und beim Aufruf der BFMAX-Methode als Obergrenze bzw. β übergeben. Diese Übergabe ähnelt der Vorgehensweise beim Alpha-Beta-Algorithmus. Die übergebenen α- bzw. β-Werte werden dann bei expandierten Knoten für frühzeitige *Cutoffs* verwendet, wie gerade oben beschrieben.

In Abb. 5.7 wird die Funktionsweise des Algorithmus grob dargestellt. Das rechte Kind ist hier der bessere Kandidat und wird als nächstes expandiert. Während des Expandierens wird der Wert des nächsten besten Knotens (4) als untere Grenze, also α, übernommen. Nach dem Expandieren wird der Wert des rechten Kindes auf das Minimum der Werte seiner Kinder, also auf 2, gesetzt. 2 ist kleiner als die untere Grenze von 4. Dadurch ist das rechte Kind nicht mehr auf dem PV-Pfad, weil der MAX-Knoten einen besseren Wert schon gefunden hatte, nämlich 4. Damit ist dieser Knoten nicht mehr für MAX-Knoten interessant und wird dadurch das linke Kind zum neuen PV-Pfad. Der Algorithmus kehrt also wieder zur Wurzel zurück und der Speicher für die Kindknoten wird freigegeben, wie in Abb. 5.7.*c* zu sehen ist. Derselbe Vorgang wird auch weiter für die anderen Knoten im Suchbaum angewendet [8].

Durch die Vorgehensweise des Algorithmus, bei der zuerst die vielversprechendsten Züge tiefer analysiert werden, erreicht BFM in den meisten Fällen die kürzeste Spielzeit im Vergleich zu anderen Algorithmen. Außerdem werden im Vergleich zu anderen Algorithmen viel weniger Knoten durch den Algorithmus bearbeitet, siehe Abb 7.2. Diese Vorgehensweise kann aber auf der anderen Seite dazu führen, dass einige Zweige des Spielbaums nicht berücksichtigt werden. Dies begrenzt die Betrachtung des Zustandsraums und kann dazu führen, dass die optimalen Züge nicht immer gefunden werden können, siehe Abb. 7.4, 7.5 und 7.6 . Dieser Effekt spiegelt sich auch in den Test-Ergebnissen wider, die in den folgenden Kapiteln durchgeführt wurden, sh. Kapitel 7.1.

Algorithm 8 Pseudocode für Rekursive-Best-First-Minimax Suche, Alg. [8]

```
1: function BFMAX(Node, Alpha, Beta)
2:     for each Child[i] of Node do
3:         M[i] := Evaluation(Child[i])
4:         if M[i] > Beta then
5:             return M[i]
6:         end if
7:     end for
8:     SORT Child[i] and M[i] in decreasing order
9:     if only one child then
10:        M[2] := -infinity
11:    end if
12:    while Alpha ≤ M[1] ≤ Beta do
13:        M[1] := BFMIN(Child[1], max(Alpha, M[2]), Beta)
14:        insert Child[1] and M[1] in sorted order
15:    end while
16:    return M[1]
17: end function
18:
19: function BFMIN(Node, Alpha, Beta)
20:    for each Child[i] of Node do
21:        M[i] := Evaluation(Child[i])
22:        if M[i] < Alpha then
23:            return M[i]
24:        end if
25:    end for
26:    SORT Child[i] and M[i] in increasing order
27:    if only one child then
28:        M[2] := infinity
29:    end if
30:    while Alpha ≤ M[1] ≤ Beta do
31:        M[1] := BFMAX(Child[1], Alpha, min(Beta, M[2]))
32:        insert Child[1] and M[1] in sorted order
33:    end while
34:    return M[1]
35: end function
```

6. Implementierung

In diesem Kapitel wird die Implementierung der im Kapitel 4 und 5 beschriebenen Evaluationsmethoden und Suchalgorithmen detailliert erläutert. Die Implementierung erfolgte in der Programmiersprache *Java* in Version 17. Zur Verwaltung der Projektabhängigkeiten, wie beispielsweise *JUnit* für Testfälle, wurde das Build-Management-Tool *Maven* verwendet. Für die Versionierung des Projekts kam das Versionskontrollsystem *Git* in Verbindung mit *GitLab* zum Einsatz. Als Entwicklungsumgebung wurde *IntelliJ IDEA 2023.1.1 (Ultimate Edition)* verwendet. Alle Tests wurden auf einem Rechner mit einem Intel Core i7, 1,5 GHz, 4-Kern-Prozessor und 12 GB RAM durchgeführt. Maximale Größe des Heap-Speichers für alle Tests war 4096 MB. Für die Dokumentation des Codes wurde Javadoc verwendet. Zur Ermittlung der CPU- und Speicherbelastung wurde JProfiler[1] eingesetzt.

Im Rahmen der Implementierung wurden die Suchalgorithmen *Alpha-Beta*, *MTD(f)* und *Best-First-Minimax* basierend auf den Konzepten, die im Kapitel 5 erläutert wurden, implementiert. Um die Leistung und Effektivität der implementierten Suchalgorithmen und Evaluationsmethoden zu bewerten, wurde ein Othello-Spiel entwickelt, das in Abb. 6.1 grob skizziert wird. Es wurde keine Benutzeroberfläche (User Interface, UI) für die Anwendung entwickelt. Stattdessen wurden mehrere Testklassen mit Hilfe von *JUnit* implementiert. Die Testfälle ermöglichten die Simulation verschiedener Szenarien und Spielsituationen, siehe Abb. 6.4, und ermöglichten die Analyse der Leistung von Algorithmen unter Verwendung unterschiedlicher Evaluationsmethoden.

Eine der wichtigsten Klassen in der Implementierung ist die *Board*-Klasse, welche die spielrelevanten Informationen enthält, siehe die Tabelle 6.1. Diese Informationen umfassen beispielsweise die Belegung der einzelnen Felder durch die Spieler, den aktuellen Spieler am Zug sowie den maximierenden Spieler und den Spielstatus. Gemäß den Regeln des Othello-Spiels beginnt immer der schwarze Spieler zum Spiel, was zur Annahme führte, dass der schwarze Spieler im Spiel immer als der maximierende Spieler betrachtet wurde.

Die zentrale Rolle in der Implementierung nimmt die *Game*-Klasse ein, welche das gesamte Spiel kontrolliert. In dieser Klasse werden die ausgewählten Algorithmen und Evaluationsmethoden für

[1] https://www.ej-technologies.com/products/jprofiler/overview.html

Methode	**Beschreibung**
findValidMoves	findet die legalen Züge für den aktuellen Spieler.
getMoveState	ist ein Zug zu einer bestimmten Position erlaubt?
makeMove	macht einen Zug zu einer bestimmten Position.
printBoard	Ausgabe des aktuellen Spielbretts und Spielers, siehe die Abb. 6.2.*a*
printBoardWithAllInformations	Ausgabe des aktuellen Spielbretts, Spielers und der legalen Züge, siehe die Abb. 6.2.*b*

Tabelle 6.1.: Wichtige Methoden der Board-Klasse

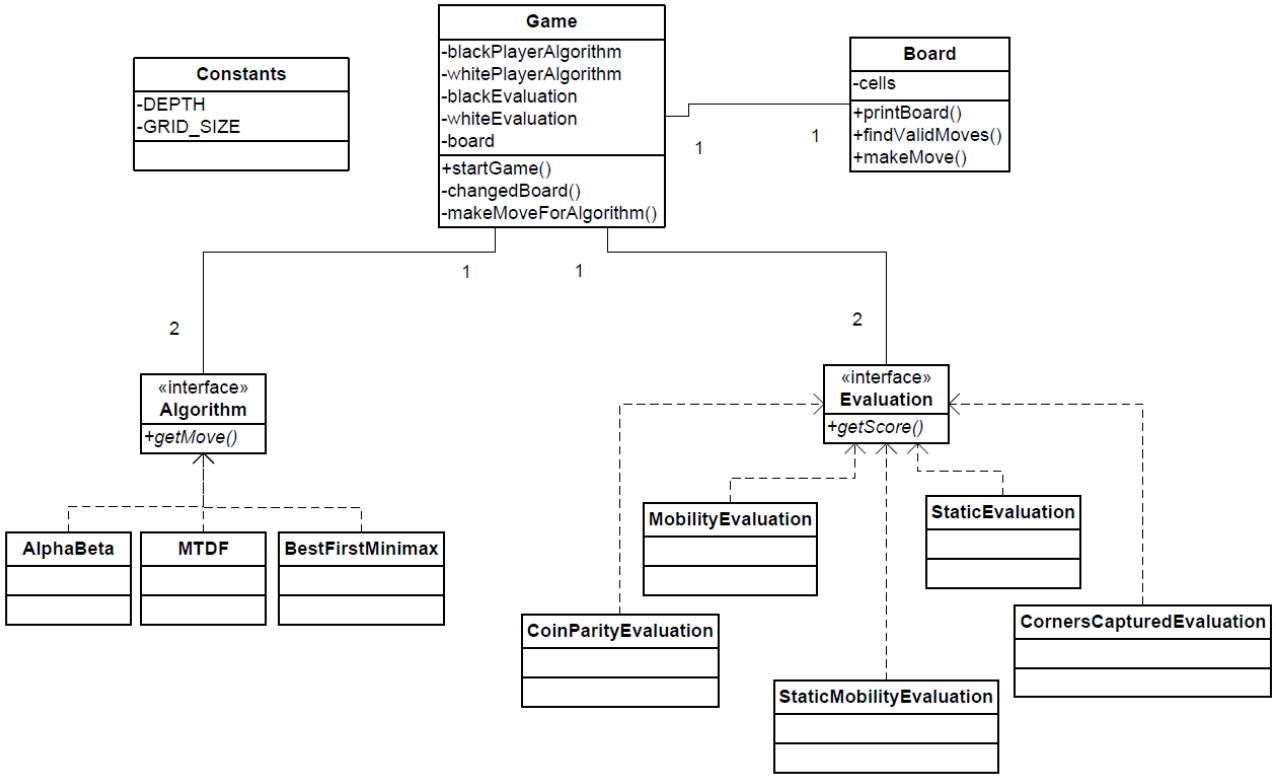


Abbildung 6.1.: Klassendiagramm für die Implementierung

Methode	Beschreibung
startGame	startet das Spiel und gibt die Kontrolle an *changedBoard*
changedBoard	führt die Züge für jeden Spieler bis zum Ende des Spiels aus. Am Ende des Spiels gibt sie den Gewinner und die Punktzahl beider Spieler aus.
makeMoveForAlgorithm	macht für einen Spieler den Zug unter Berücksichtigung des ausgewählten Algorithmus und der Evaluationsmethoden.

Tabelle 6.2.: Wichtige Methoden der Game-Klasse

beide Spieler verwaltet. Das Spiel wird über die Methode *startGame* gestartet, welche dann die Kontrolle an die Methode *changedBoard* übergibt. Letztere wird über einen *Change Listener* nach jedem Zug aufgerufen, bis das Spiel beendet ist. Die Aufgabe von *changedBoard* besteht darin, für jeden Spieler den nächsten besten Zug basierend auf seinem ausgewählten Algorithmus und den Evaluationsmethoden beider Spieler zu ermitteln und auszuführen. Nach Beendigung des Spiels gibt die Methode den Gewinner sowie die Punktzahlen beider Spieler aus, wie in Tabelle 6.2 dargestellt.

Um die Austauschbarkeit von Algorithmen und Evaluationsmethoden innerhalb der Applikation und Testklassen zu erleichtern, wurden diese so entwickelt, dass sie jeweils ein bestimmtes Interface implementieren: *Algorithm* bzw. *Evaluation*, siehe die Abb. 6.1.

Zur zentralen Verwaltung der Einstellungen wurde eine Klasse namens *Constants* erstellt. Diese Klasse enthält statische Konstanten, die von allen Klassen der Applikation verwendet werden, unter anderem *DEPTH* für die Tiefe von Suchalgorithmen und *GRID_SIZE* für die Größe des Spielbretts.

(a)

```
Aktueller Spieler ist: BLACK

    Y0  Y1  Y2  Y3  Y4  Y5  Y6  Y7
X0  -   -   -   -   -   -   -   -
X1  -   -   -   -   -   -   -   -
X2  -   -   -   -   -   -   -   -
X3  -   -   -   W   B   -   -   -
X4  -   -   -   B   W   -   -   -
X5  -   -   -   -   -   -   -   -
X6  -   -   -   -   -   -   -   -
X7  -   -   -   -   -   -   -   -
```

(b)

```
Aktueller Spieler ist: BLACK

Black-Score ist: 2
WHITE-Score ist: 2

Valide Züge sind:
------------------
Position{x=2, y=3}
Position{x=3, y=2}
Position{x=4, y=5}
Position{x=5, y=4}

    Y0  Y1  Y2  Y3  Y4  Y5  Y6  Y7
X0  -   -   -   -   -   -   -   -
X1  -   -   -   -   -   -   -   -
X2  -   -   -   X   -   -   -   -
X3  -   -   X   W   B   -   -   -
X4  -   -   -   B   W   X   -   -
X5  -   -   -   -   X   -   -   -
X6  -   -   -   -   -   -   -   -
X7  -   -   -   -   -   -   -   -
```

Abbildung 6.2.: *(a)* Ausgabe für die *printBoard*-Methode. Aktueller Spieler und aktuelles Spielbrett wird ausgegeben. *(b)* Ausgabe für die *printBoardWithAllInformations*-Methode. Hier werden noch ergänzend die Punkte bzw. Scores für jeden Spieler und die legalen Züge für den aktuellen Spieler ausgegeben. Die legalen Züge im Spielbrett sind mit einem *X* gekennzeichnet.

6.1. Erstellung von Testfällen

Für die Erstellung von Testfällen wurden Online-Spiele verwendet, auf denen verschiedene reale Spielsituationen erstellt wurden[2]. Die Spiele wurden bis zu einem bestimmten Spielstand gespielt und das Ergebnis wurde in einer Textdatei namens *spiel.txt* festgehalten. Mithilfe einer selbst erstellten Hilfsklasse namens *TestCreator* wurde diese Datei ausgelesen. Dadurch konnten die Felder auf dem Spielbrett in den Testmethoden schneller gesetzt werden, was die Arbeit bei vielen Tests erheblich erleichterte, wie in Abb. 6.3 dargestellt. Auf diese Weise wurden 20 Testfälle erstellt, die in Abb. 6.4 zu finden sind und bei allen Tests verwendet wurden.

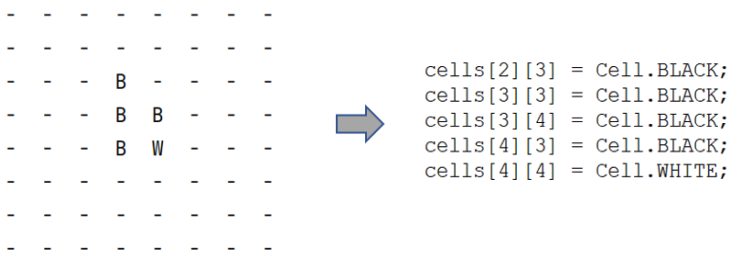


Abbildung 6.3.: Erstellung von Testfällen mit Hilfe des *TestCreator*s

[2]Zum Beispiel: https://www.bernhard-gaul.de/spiele/reversi/reversi.php (abgerufen am 09.09.2023)

Klasse	Beschreibung
AlphaBetaTest	Berechnung des besten Zugs für jedes Spiel aus Abb. 6.4 mit AlphaBeta
MTDFTest	Berechnung des besten Zugs für jedes Spiel aus Abb. 6.4 mit MTDF
BestFirstMinimaxTest	Berechnung des besten Zugs für jedes Spiel aus Abb. 6.4 mit BestFirstMinimax
CoinParityEvaluationTest	Berechnung des Evaluationswertes für jedes Spiel aus Abb. 6.4 mit CoinParityEvaluation
CornersCapturedEvaluationTest	Berechnung des Evaluationswertes für jedes Spiel aus Abb. 6.4 mit CornersCapturedEvaluation
MobilityEvaluationTest	Berechnung des Evaluationswertes für jedes Spiel aus Abb. 6.4 mit MobilityEvaluation
StaticEvaluationTest	Berechnung des Evaluationswertes für jedes Spiel aus Abb. 6.4 mit StaticEvaluation
StaticMobilityEvaluationTest	Berechnung des Evaluationswertes für jedes Spiel aus Abb. 6.4 mit StaticMobilityEvaluation
AllGamesTest	Test-Klasse, in der Algorithmen paarweise miteinander verglichen werden (Ermittlung des Gewinners) und zwar für alle Spiele aus Abb. 6.4
MoveSpeedTest	Berechnung der Zeit, um den besten Zug herauszufinden und zwar für alle Spiele aus Abb. 6.4
ZobristHashTest	Tests bezüglich des Zobrist-Hashing

Tabelle 6.3.: Wichtige Test-Klassen

Die Test-Klassen aus Tabelle 6.3 wurden für alle Tests in dieser vorliegenden Arbeit verwendet. Dabei haben alle Test-Klassen die Spiele aus Abb. 6.4 berücksichtigt.

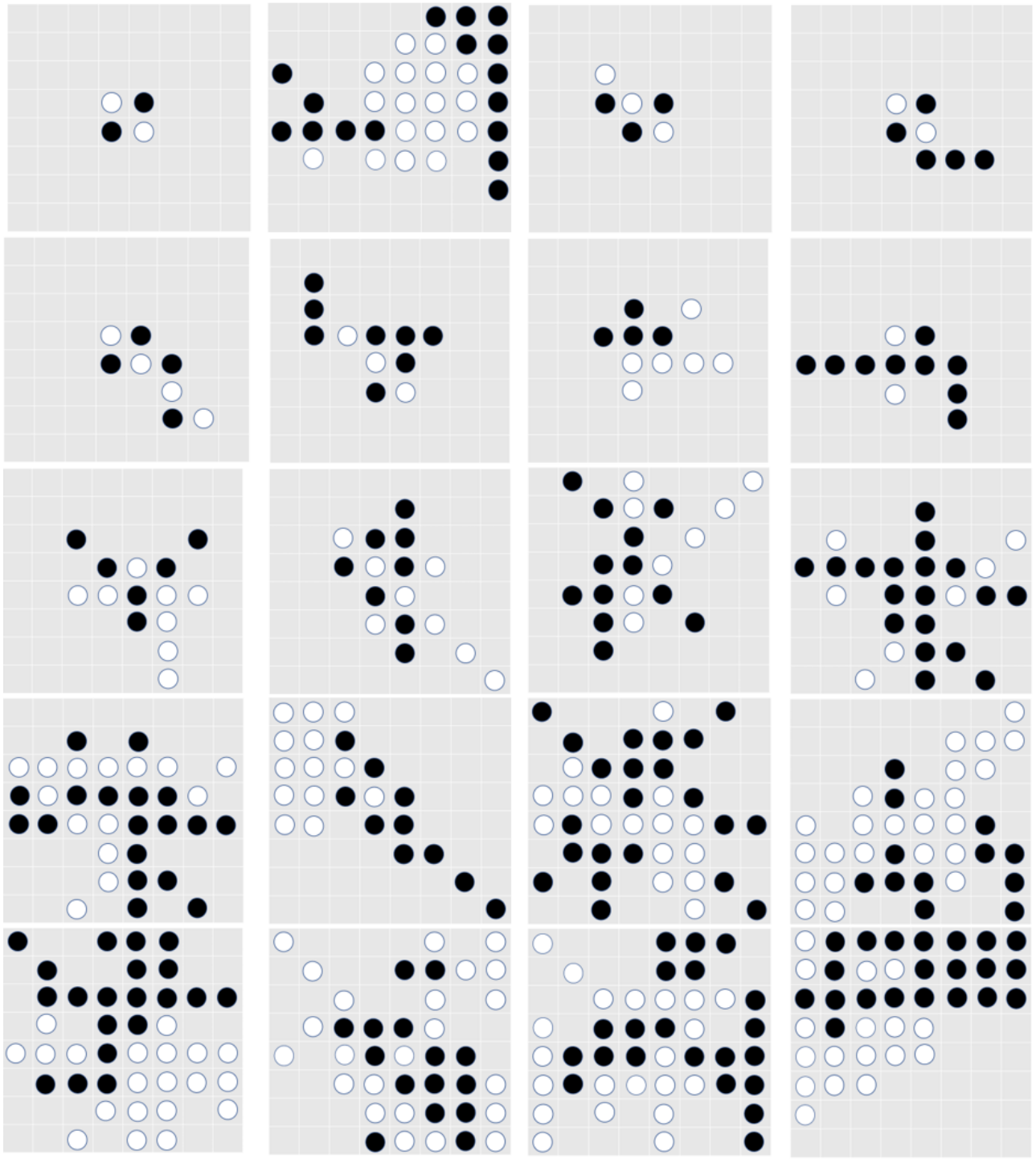


Abbildung 6.4.: Erstellung von Testfällen. Für den Vergleich von Algorithmen bzw. Evaluationsmethoden 20 unterschiedliche Testfälle erstellt, die bei allen Tests verwendet wurden. Diese Testfälle werden von links nach rechts durchnummeriert und werden im Kapitel 7 per *TF-01*, ..., *TF-20* referenziert.

7. Ergebnisse

7.1. Vergleich von Algorithmen

Es wurden verschiedene Strategien verwendet, um die Algorithmen miteinander zu vergleichen.

Bei der ersten Strategie wurde eines der zuvor genannten Spiele, *TF-01* aus Abb. 6.4, ausgewählt. Das Spiel wurde mit den Algorithmen paarweise bis zum Ende des Spiels gespielt, wobei für jeden Algorithmus dieselbe Evaluationsmethode mit statischen Gewichten verwendet wurde, siehe Kapitel 4.1. Das Spiel wurde in verschiedenen Tiefen (5 bis 10) gespielt, wobei für jede Tiefe zwei Durchläufe durchgeführt wurden. In einem Durchlauf begann Algorithmus 1 und im anderen Durchlauf Algorithmus 2. Das Ziel dieser Strategie war es, die durchschnittliche Zeit, die die Algorithmen bis zum Ende des Spiels benötigten, sowie die Anzahl der Knoten, die bearbeitet wurden, zu ermitteln. Diese Messungen können zur Berechnung der Leistung und Effizienz der Algorithmen herangezogen werden.

Alle Ergebnisse sind in Abb. 7.1 zusammengefasst. In der Abb. sind die Algorithmen, die im Durchlauf als Erster gestartet haben, mit einem blauen Hintergrund markiert. In Abb. 7.2 werden diese Daten in einer übersichtlicheren Form dargestellt. Unter Berücksichtigung dieser Informationen ergeben sich die folgenden Ergebnisse:

- BFM erreicht in den meisten Fällen die kürzeste Spielzeit, gefolgt von MTD(f) und AB.
- AB bearbeitet in den meisten Fällen die höchste Anzahl von Knoten, gefolgt von MTD(f) und BFM.
- Ab der Tiefe 8 ist bei AB ein enormer Anstieg sowohl der benötigten Zeit als auch der Anzahl der Knoten zu beobachten.
- Mit zunehmender Tiefe wird MTD(f) im Vergleich zu AB immer besser, benötigt weniger Zeit und wertet weniger Knoten aus. Diese Ergebnisse bestätigen damit auch die Ergebnisse aus Tommy et. al [17].

Bei der zweiten Strategie wurde getestet, wie schnell die Algorithmen den besten Zug in jeder Tiefe finden konnten. Dazu wurden die Algorithmen in 20 verschiedenen Spielen, siehe Abb. 6.4, miteinander verglichen, wobei die Suchtiefen von 5 bis 10 verwendet wurden, um ihre Geschwindigkeit bei der Berechnung des optimalen Zuges in diesen Spielsituationen zu messen. Die Ergebnisse dieser Messungen sind in Abb. 7.3 dargestellt. Wie erwartet zeigte der BFM-Algorithmus eine fast lineare Leistung, was auf den Entscheidungsprozess bei der Auswahl des besten Kandidaten in der BFM-Suche zurückzuführen ist. AB und MTD(f) hingegen zeigen ein ähnliches Verhalten, wobei MTD(f) im Vergleich zu AB bessere Ergebnisse erzielte.

Bei der dritten Strategie wurde getestet, welcher Algorithmus, in welcher Tiefe, welche Positionen als bester Zug zurücklieferte. Dafür wurden die Algorithmen erneut in 20 verschiedenen Spielen getestet,

siehe Abb. 6.4, und bei unterschiedlichen Suchtiefen (5 bis 10) ausgewertet. Dabei wurde für jeden Algorithmus dieselbe Evaluationsmethode verwendet, siehe Kapitel 4.1. Wie aus den Abbildungen 7.4, 7.5 und 7.6 ersichtlich ist, haben die beiden ersten Minimax-Varianten *AB* und *MTD(f)* wie erwartet dieselben Ergebnisse erzielt. Nur im Spiel 14 haben die beiden Algorithmen für die Tiefen 8 und 10 jeweils umgekehrt entschieden, siehe die gelb markierten Felder in den Abbildungen 7.4 und 7.5. Die Positionen, bei denen die Algorithmen zu unterschiedlichen Entscheidungen gekommen sind, wurden durch Evaluationsmethoden gleichwertig bewertet, was zu diesem Ergebnis geführt hat, für eine detaillierte Beschreibung dieses Phänomens siehe etwas unten. Wie bereits erwähnt, handelt es sich bei BFM um eine Variante des Minimax-Algorithmus, bei der eine Best-First-Suche verwendet wird, um den vielversprechendsten Teilbaum des Spielbaums zu erkunden. Anstatt den gesamten Spielbaum in einer gleichmäßigen Tiefe zu durchsuchen, werden nur ausgewählte vielversprechende Zweige tiefer analysiert. BFM wählt also die vielversprechendsten Züge aus, um sie weiter zu untersuchen. Dadurch werden einige Zweige des Spielbaums nicht berücksichtigt, was zu einer begrenzten Betrachtung des Zustandsraums führt. Dieser Effekt spiegelt sich auch in den Ergebnissen wider, da BFM bei einer fest vorgegebenen Tiefe im Gegensatz zu AB und MTD(f) nicht immer den besten Zug im Sinne des Minimax-Werts finden kann. Aus diesem Grund unterscheiden sich die Ergebnisse des BFM-Algorithmus deutlich von den Ergebnissen beider anderer Algorithmen.

Im Rahmen der vierten Strategie wurde untersucht, welcher Algorithmus unter unbegrenzter Zeitvorgabe bessere Ergebnisse erzielt. Alle Ergebnisse sind im Anhang B aufgeführt. Für den Vergleich wurden 20 verschiedene Spiele verwendet, siehe Abb. 6.4. Die Spiele wurden in verschiedenen Tiefen (5 bis 10) gespielt, wobei für jede Tiefe zwei Durchläufe durchgeführt wurden. In einem Durchlauf begann Algorithmus 1 und im anderen Durchlauf Algorithmus 2. Während dieser Tests gab es keine Zeitbeschränkungen für die Züge der Algorithmen. Allerdings stellte sich heraus, dass Spiele ab der Tiefe 10 entweder sehr schwierig zu spielen waren oder aufgrund von *Out of Memory (OOM)*-Fehlern nicht erfolgreich beendet werden konnten. Daher wurden nicht alle Ergebnisse der Tiefe 10 in die Auswertung einbezogen. Eine tabellarische Darstellung aller Testergebnisse befindet sich in Abb. 7.7. In der Tabelle sind die Anzahl der gewonnenen Spiele pro Algorithmus in Bezug auf insgesamt 40 Spiele aufgeführt.

Basierend auf diesen Informationen ergeben sich die folgenden Ergebnisse:

- Die unerwartetste Beobachtung ergab sich aus den Partien zwischen AB und MTD(f), wie in Kapitel B.2 detailliert zu sehen ist. Obwohl vorherige Tests zeigten, dass AB und MTD(f) bei der Suche nach dem besten Zug übereinstimmende Ergebnisse liefern, siehe 7.4 und 7.5, ergaben sich hier Fälle, in denen beide Algorithmen während eines Spiels zu unterschiedlichen Ergebnissen führten. Es wurde erwartet, dass die Spiele unabhängig vom Startspieler mit AB oder MTD(f) immer zu denselben Ergebnissen führen. Die Analyse zeigte jedoch, dass die Algorithmen nach einigen Zügen unterschiedliche Positionen als besten Zug wählten, wodurch sich das Endergebnis des Spiels änderte. Ein Beispiel dafür wird in Abbildung 7.8 für den *TF-01* aus Abbildung 6.4 veranschaulicht. Wie es in der Abb. 7.8 zu sehen ist, entscheidet hier MTD(f) für die Position (2,4), währenddessen AB für die Position (3,5) entscheidet. Weitere Analyse haben gezeigt, dass diese beiden Positionen durch Evaluationsmethoden gleich bewertet wurden. Das hat dazu geführt, dass ein Algorithmus für eine und der andere Algorithmus dann für andere Position entschieden hat. Dieses Phänomen ist unter dem Namen *Tie-Breaking* bekannt. Wenn zwei oder mehr Elemente in einem Algorithmus die gleiche Bewertung oder Priorität haben, muss das Tie-Breaking angewendet werden, um eine eindeutige Entscheidung

zu treffen. Das Ergebnis zeigt eindeutig, dass sich diese Algorithmen bei solchen Fällen anders entscheiden.

- Obwohl der AB-Algorithmus in mehr Spielen gegen den MTD(f)-Algorithmus gewonnen hat, kann dennoch gesagt werden, dass der MTD(f)-Algorithmus aufgrund der Informationen aus früheren Tests als überlegen angesehen werden kann. Der MTD(f)-Algorithmus zeichnet sich durch kürzere Spielzeiten und die Bearbeitung einer geringeren Anzahl von Knoten aus. Bei AB hingegen gibt es ab der Tiefe 8 einen enormen Anstieg der zu bearbeitenden Knoten, wodurch mehrere Spiele in der Tiefe 10 sehr lange gedauert haben.

Diese Ergebnisse wurden ebenfalls durch verschiedene Tests mit *JProfiler* bestätigt, wie im Anhang D zu sehen ist. Dazu wurde der Testfall *TF-09* mit verschiedenen Algorithmen nur in der Tiefe 10 paarweise gespielt. In allen Tests verwendeten die Algorithmen die gleiche Evaluationsmethode mit statischen Gewichten, siehe Kapitel 4.1. Aus diesen Ergebnissen lassen sich folgende Erkenntnisse ableiten:

- Selbst wenn der AB-Algorithmus ab Tiefe 8 enorme Anzahl von Knoten zu verarbeiten hat, zeigt die Partie *AB vs. AB*, siehe Abb. D.1, dass der Speicherverbrauch trotzdem stabil bleibt und nicht zu einem *Out of Memory (OOM)* Exception führt, obwohl das Spiel über 30 Minuten dauerte.
- Bei den Partien, an denen *MTD(f)* beteiligt war, war der Speicherverbrauch außerordentlich hoch, wie aus den Abbildungen D.2, D.4 und D.5 ersichtlich ist. Obwohl der MTDF-Algorithmus schnell ist, ist der Speicherverbrauch aufgrund der Verwendung von Transpositionstabellen recht hoch. Mehrere Spiele für die Tiefe 10 aus Anhang B wurden genau aus diesem hohen Speicherverbrauch mit einer *Out Of Memory* Exception beendet, an denen der MTD(f)-Algorithmus beteiligt war. Es ist zu beachten, dass in dieser vorliegenden Arbeit Transpositionstabellen ohne eine festgelegte Größe verwendet wurden.

Nach Auswertung sämtlicher Testergebnisse können folgende Schlussfolgerungen gezogen werden. Obwohl BFM die kürzeste Spielzeit aufweist und weniger Knoten als AB und MTD(f) bewertet, erweisen sich AB und MTD(f) im Vergleich zu BFM als überlegen. Außerdem kann BFM aufgrund der begrenzten Tiefensuche und des möglichen Ausschlusses von Pfaden durch die Sortierung vielversprechender Züge nicht immer den optimalen Zug finden. Der MTD(f)-Algorithmus zeichnet sich durch die Fähigkeit aus, kürzere Spielzeiten zu erzielen und dabei effizient eine reduzierte Anzahl von Knoten zu verarbeiten. Somit erweist sich MTD(f) im Vergleich zu AB als überlegen. Allerdings muss dabei der hohe Speicherverbrauch durch die Transpositionstabellen in Kauf genommen werden.

7.2. Vergleich von Evaluationsmethoden

Für den Vergleich der Evaluationsmethoden wurden fünf Testfälle aus Abb. 6.4 ausgewählt (*TF-01*, *TF-09*, *TF-12*, *TF-14* und *TF-20*), siehe Abb. 7.9. Beide Spieler verwendeten den Suchalgorithmus *MTD(f)* aus Kapitel 5.2, wobei die Tiefe der Algorithmen konstant auf 8 festgelegt war. Die im Kapitel 4 behandelten fünf Evaluationsmethoden wurden paarweise miteinander verglichen, wobei für jedes Spiel zwei Durchläufe durchgeführt wurden. In einem Durchlauf verwendete Algorithmus-1 die erste Evaluationsmethode, während im anderen Durchlauf Algorithmus-2 sie verwendete. Im Anhang C sind alle Ergebnisse tabellarisch dargestellt. Wie die Ergebnisse zeigen, wurden für die ausgewählten

fünf Testfälle jeweils zwei Durchläufe gespielt und die Ergebnisse in den Spalten 2 und 3 aufgezeichnet. In den Spalten 4 und 5 wurde außerdem vermerkt, wie viele Spielsteine jeder Spieler insgesamt in der jeweiligen Tiefe gewonnen hat, was beim Vergleich der Evaluationsmethoden eine wichtige Rolle spielte.

Die Anzahl der Spiele, die jede Evaluationsmethode in den paarweisen Vergleichen und insgesamt gewonnen hat, wurde in Abb. 7.10 zusammengefasst. In der Abbildung bedeutet *[a-b]* in einer Zelle, dass die Evaluationsmethode in Spalte 1 *a*-mal und die Evaluationsmethode in Spalte 2 *b*-mal gewonnen hat. Spiele, die unentschieden endeten, wurden bei der Berechnung ignoriert. Lediglich ein Spiel zwischen *Mobility* und *StaticMobility* endete unentschieden und wurde daher ignoriert (die gelb markierte Zelle). Aus diesem Grund wurden insgesamt nicht 100, sondern 99 Spiele in die Berechnung einbezogen.

Die vorliegenden Ergebnisse bekräftigen die Befunde der Studie von Sannidhanam et al. [15]. Ähnlich wie in ihrer Arbeit zeigt sich auch hier, dass die Evaluationsmethode *CornersCaptured* im Vergleich zu *Mobility* und *CoinParity* überlegen ist. Es ist jedoch zu beachten, dass sie in ihrer Arbeit die Evaluationsmethode mit statischen Gewichten, wie im Kapitel 4.1 beschrieben, nicht berücksichtigt haben. Die vorliegenden Ergebnisse zeigen, dass die Evaluationsmethode mit statischen Gewichten sogar besser abschneidet als *CornersCaptured*. Damit untermauern die vorliegenden Ergebnisse auch die Ergebnisse von Barber et al. [3], wo sich die Evaluationsmethode mit statischen Gewichten (in der Arbeit wird die Methode aber unter einem anderem Namen *Weight Matrix* beschrieben) als die erfolgreichste Methode erwiesen hat. Besonders bemerkenswert war die Mischmethode *StaticMobility*, die sich als äußerst erfolgreich erweist und mit 29 Siegen den ersten Platz in Abb. 7.10 belegte.

	Durchschnittliche Zeit	
Tiefe	AB	BFM
5	0,16	0,02
5	0,24	0,03
6	0,58	0,05
6	0,76	0,05
7	18,17	0,07
7	3,86	0,08
8	8,72	0,12
8	6,90	0,13
9	59,43	0,21
9	64,00	0,20
10	113,77	0,41
10	142,74	0,45
	69,89	0,15

Durchschnittliche Anzahl an Knoten	
AB	BFM
1677435	35131
3092081	52751
8098009	177539
10589084	133599
462450950	267362
79023056	304714
193055210	580999
163323216	608748
1278483836	1179442
1645672359	913378
2884246593	2297188
3480520799	2511560
850852719	755201

	Durchschnittliche Zeit	
Tiefe	BFM	MTD(f)
5	0,03	0,20
5	0,02	0,13
6	0,06	0,66
6	0,07	0,43
7	0,07	1,89
7	0,10	2,24
8	0,13	1,15
8	0,16	7,21
9	0,18	17,99
9	0,17	9,37
10	0,45	18,26
10	0,39	19,78
	0,15	6,61

Durchschnittliche Anzahl an Knoten	
BFM	MTD(f)
48563	293939
32834	149074
148504	991650
145160	525810
235379	4137389
339155	3103634
532574	2973318
620259	12963345
885338	34746596
794656	20650932
2164086	53492519
2154283	40683232
675066	14559287

	Durchschnittliche Zeit	
Tiefe	AB	MTD(f)
5	0,13	0,08
5	0,18	0,12
6	0,74	0,31
6	0,54	0,38
7	10,30	2,23
7	2,05	1,28
8	7,15	1,45
8	5,35	1,30
9	163,68	23,85
9	102,49	24,64
10	413,84	129,55
10	123,72	11,86
	69,18	16,42

Durchschnittliche Anzahl an Knoten	
AB	MTD(f)
1369198	118836
2221203	173484
10239216	669261
9556002	994955
223412786	4419257
34895630	2633238
160166118	3764885
105397974	2956621
3991768498	55953140
2484891975	49596235
6519584169	172779828
2454677899	26110778
1333181722	26680877

Abbildung 7.1.: Vergleich von Algorithmen in unterschiedlichen Tiefen (5 bis 10) für den TF-01 aus der Abb. 6.4. Bei jeder Tiefe werden zwei Durchläufe durchgeführt. Algorithmen mit blauem Hintergrund starten zuerst.

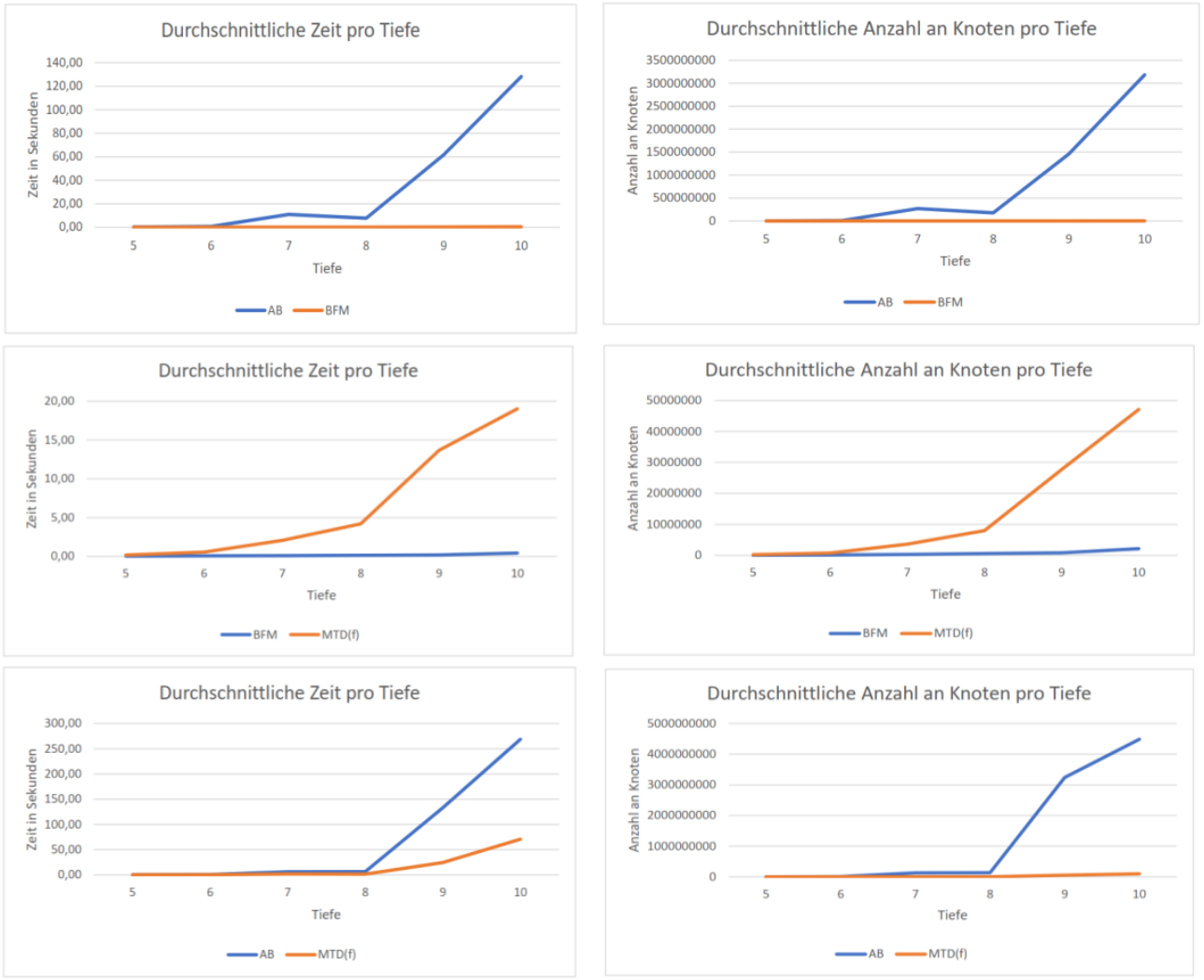


Abbildung 7.2.: Vergleich von Algorithmen in unterschiedlichen Tiefen (5 bis 10) für den TF-01 aus der Abb. 6.4 in einer übersichtlicheren Form. Bei jeder Tiefe werden zwei Durchläufe durchgeführt.

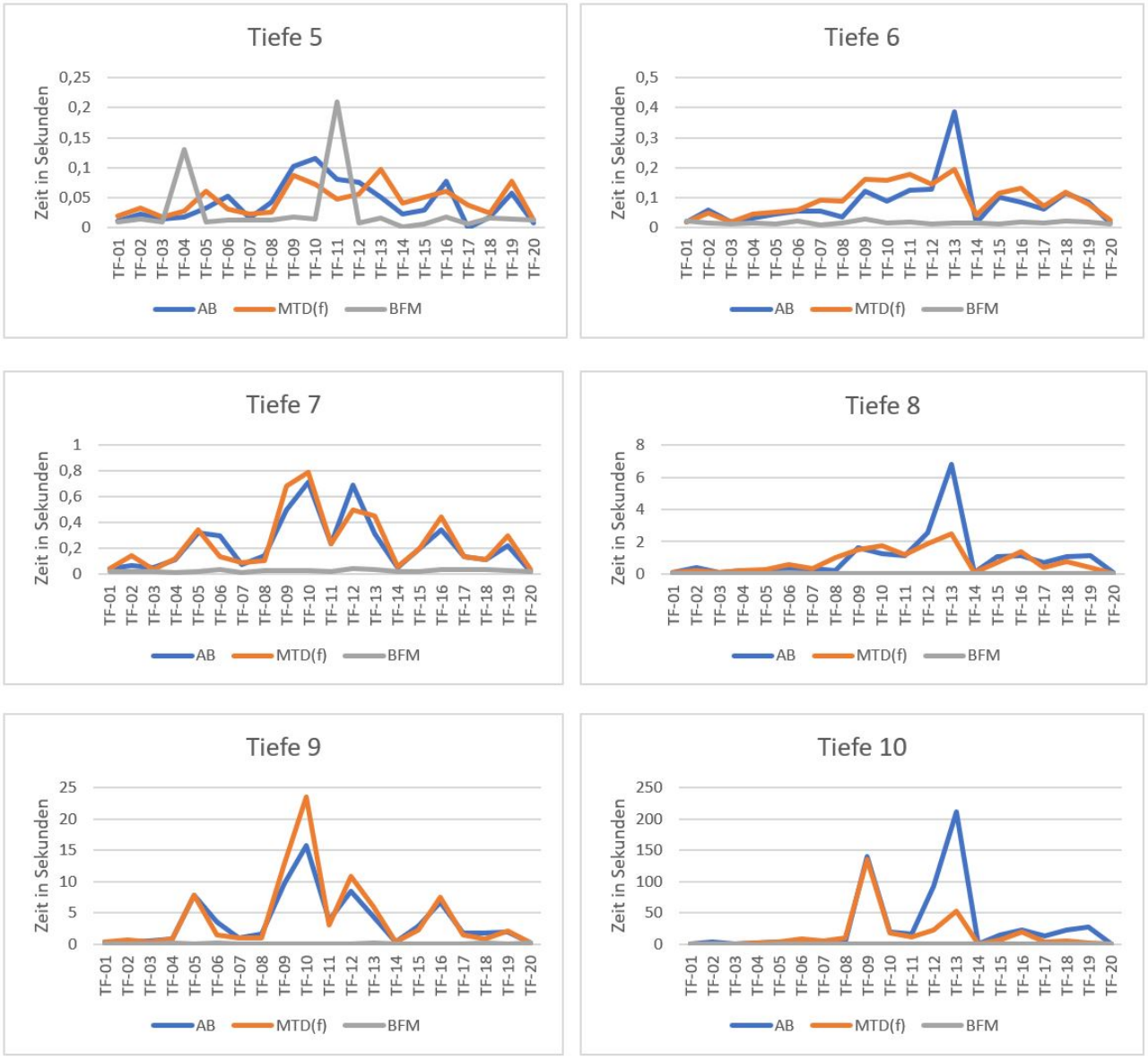


Abbildung 7.3.: Wie schnell ermitteln die Algorithmen den besten Zug in jeder Tiefe. Um alle Ergebnisse tabellarisch zu sehen, siehe die Tabellen im Anhang A.

AB	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
TF-02	(3,2)	(3,2)	(3,2)	(3,2)	(3,2)	(3,2)
TF-03	(2,3)	(1,2)	(2,3)	(1,2)	(2,3)	(2,3)
TF-04	(5,3)	(5,3)	(4,2)	(5,3)	(5,3)	(5,3)
TF-05	(6,7)	(5,4)	(6,7)	(5,4)	(6,7)	(5,4)
TF-06	(6,0)	(2,2)	(6,0)	(2,2)	(6,0)	(5,4)
TF-07	(1,6)	(5,2)	(1,6)	(5,2)	(1,6)	(5,2)
TF-08	(5,1)	(3,1)	(5,1)	(3,1)	(3,5)	(3,5)
TF-09	(7,6)	(5,7)	(7,6)	(4,7)	(7,6)	(5,1)
TF-10	(4,5)	(4,6)	(4,5)	(4,6)	(4,5)	(4,6)
TF-11	(6,3)	(5,4)	(6,3)	(3,6)	(6,3)	(5,4)
TF-12	(4,2)	(2,3)	(4,2)	(2,3)	(4,2)	(2,3)
TF-13	(2,6)	(7,3)	(2,6)	(7,3)	(6,2)	(7,3)
TF-14	(4,2)	(0,3)	(0,3)	(4,2)	(0,3)	(0,3)
TF-15	(2,0)	(2,5)	(6,3)	(5,6)	(6,3)	(2,5)
TF-16	(3,6)	(2,2)	(3,6)	(2,2)	(3,6)	(2,2)
TF-17	(3,2)	(2,0)	(7,6)	(7,3)	(7,6)	(7,3)
TF-18	(1,2)	(7,2)	(4,1)	(7,2)	(4,1)	(7,2)
TF-19	(6,6)	(4,6)	(6,6)	(4,6)	(6,6)	(1,3)
TF-20	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)

Abbildung 7.4.: Die besten Züge für unterschiedliche Spiele und Tiefen bei AB

MTD(f)	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
TF-02	(3,2)	(3,2)	(3,2)	(3,2)	(3,2)	(3,2)
TF-03	(2,3)	(1,2)	(2,3)	(1,2)	(2,3)	(2,3)
TF-04	(5,3)	(5,3)	(4,2)	(5,3)	(5,3)	(5,3)
TF-05	(6,7)	(5,4)	(6,7)	(5,4)	(6,7)	(5,4)
TF-06	(6,0)	(2,2)	(6,0)	(2,2)	(6,0)	(5,4)
TF-07	(1,6)	(5,2)	(1,6)	(5,2)	(1,6)	(5,2)
TF-08	(5,1)	(3,1)	(5,1)	(3,1)	(3,5)	(3,5)
TF-09	(7,6)	(5,7)	(7,6)	(4,7)	(7,6)	(5,1)
TF-10	(4,5)	(4,6)	(4,5)	(4,6)	(4,5)	(4,6)
TF-11	(6,3)	(5,4)	(6,3)	(3,6)	(6,3)	(5,4)
TF-12	(4,2)	(2,3)	(4,2)	(2,3)	(4,2)	(2,3)
TF-13	(2,6)	(7,3)	(2,6)	(7,3)	(6,2)	(7,3)
TF-14	(4,2)	(0,3)	(0,3)	(0,3)	(0,3)	(4,2)
TF-15	(2,0)	(2,5)	(6,3)	(5,6)	(6,3)	(2,5)
TF-16	(3,6)	(2,2)	(3,6)	(2,2)	(3,6)	(2,2)
TF-17	(3,2)	(2,0)	(7,6)	(7,3)	(7,6)	(7,3)
TF-18	(1,2)	(7,2)	(4,1)	(7,2)	(4,1)	(7,2)
TF-19	(6,6)	(4,6)	(6,6)	(4,6)	(6,6)	(1,3)
TF-20	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)

Abbildung 7.5.: Die besten Züge für unterschiedliche Spiele und Tiefen bei MTD(f)

BFMM	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	(4,5)	(4,5)	(4,5)	(4,5)	(4,5)	(4,5)
TF-02	(5,2)	(5,2)	(5,2)	(5,2)	(5,2)	(5,2)
TF-03	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)
TF-04	(2,4)	(2,4)	(2,4)	(3,5)	(3,5)	(3,5)
TF-05	(5,4)	(5,4)	(5,4)	(5,4)	(5,4)	(5,4)
TF-06	(6,0)	(6,0)	(6,0)	(6,0)	(6,0)	(6,0)
TF-07	(6,3)	(6,3)	(6,3)	(6,3)	(6,3)	(6,3)
TF-08	(3,1)	(3,1)	(3,1)	(3,1)	(3,1)	(3,1)
TF-09	(5,6)	(5,6)	(5,6)	(5,6)	(5,6)	(5,6)
TF-10	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)
TF-11	(3,6)	(3,6)	(3,6)	(3,6)	(3,6)	(3,6)
TF-12	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)	(2,5)
TF-13	(1,3)	(1,3)	(1,3)	(1,3)	(1,3)	(1,3)
TF-14	(2,4)	(2,4)	(0,3)	(0,3)	(0,3)	(0,3)
TF-15	(6,3)	(6,3)	(6,3)	(6,3)	(6,3)	(6,3)
TF-16	(4,1)	(4,1)	(4,1)	(4,1)	(4,1)	(4,1)
TF-17	(6,6)	(6,6)	(6,6)	(6,6)	(6,6)	(6,6)
TF-18	(3,6)	(3,6)	(3,6)	(3,6)	(3,6)	(3,6)
TF-19	(6,3)	(6,3)	(6,3)	(6,3)	(6,3)	(6,3)
TF-20	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)

Abbildung 7.6.: Die besten Züge für unterschiedliche Spiele und Tiefen bei BFM

Tiefe	AB	MTD(f)	Unentschieden	OOM
5	17	22	1	0
6	27	12	1	0
7	20	20	0	0
8	24	16	0	0
9	17	21	2	0
10	21	11	1	7
	126	102	4	7

Tiefe	AB	BFM	Unentschieden	OOM
5	20	19	1	0
6	20	18	2	0
7	15	25	0	0
8	29	11	0	0
9	10	27	3	0
10	34	6	0	0
	128	106	6	0

Tiefe	MTD(f)	BFM	Unentschieden	OOM
5	16	24	0	0
6	24	16	0	0
7	19	21	0	0
8	30	9	1	0
9	17	22	1	0
10	24	11	1	4
	130	103	3	4

Abbildung 7.7.: Ergebnisse beim Vergleich von Algorithmen ohne Zeitlimit. Hier wurde es gemessen, welcher Algorithmus wie viele Spiele von insgesamt 20 Test-Spielen aus der Abb. 6.4 ohne ein Zeitlimit gewonnen hat. Die Spiele wurden in verschiedenen Tiefen (5 bis 10) gespielt, wobei für jede Tiefe zwei Durchläufe durchgeführt wurden. In einem Durchlauf begann Algorithmus 1 und im anderen Durchlauf Algorithmus 2. In der Tabelle sind damit die Anzahl der gewonnenen Spiele pro Algorithmus in Bezug auf insgesamt 40 Spiele aufgeführt. OOM steht für Out-Of-Memory.

```
SCHWARZ = MTDF

Aktueller Spieler ist: BLACK

Black-Score ist: 2
WHITE-Score ist: 2

Valide Züge sind:
--------------------
Position{x=2, y=3}
Position{x=3, y=2}
Position{x=4, y=5}
Position{x=5, y=4}

    Y0  Y1  Y2  Y3  Y4  Y5  Y6  Y7
X0  -   -   -   -   -   -   -   -
X1  -   -   -   -   -   -   -   -
X2  -   -   -   X   -   -   -   -
X3  -   -   X   W   B   -   -   -
X4  -   -   -   B   W   X   -   -
X5  -   -   -   -   X   -   -   -
X6  -   -   -   -   -   -   -   -
X7  -   -   -   -   -   -   -   -

Spieler BLACK hat gespielt: Position{x=2, y=3}
-----------------------------------------------
Spieler WHITE hat gespielt: Position{x=2, y=2}
-----------------------------------------------
Spieler BLACK hat gespielt: Position{x=2, y=1}
-----------------------------------------------
Spieler WHITE hat gespielt: Position{x=2, y=4}
```

```
SCHWARZ = AB

Aktueller Spieler ist: BLACK

Black-Score ist: 2
WHITE-Score ist: 2

Valide Züge sind:
--------------------
Position{x=2, y=3}
Position{x=3, y=2}
Position{x=4, y=5}
Position{x=5, y=4}

    Y0  Y1  Y2  Y3  Y4  Y5  Y6  Y7
X0  -   -   -   -   -   -   -   -
X1  -   -   -   -   -   -   -   -
X2  -   -   -   X   -   -   -   -
X3  -   -   X   W   B   -   -   -
X4  -   -   -   B   W   X   -   -
X5  -   -   -   -   X   -   -   -
X6  -   -   -   -   -   -   -   -
X7  -   -   -   -   -   -   -   -

Spieler BLACK hat gespielt: Position{x=2, y=3}
-----------------------------------------------
Spieler WHITE hat gespielt: Position{x=2, y=2}
-----------------------------------------------
Spieler BLACK hat gespielt: Position{x=2, y=1}
-----------------------------------------------
Spieler WHITE hat gespielt: Position{x=3, y=5}
```

Abbildung 7.8.: *Tie Breaking*: Es wurde erwartet, dass die Spiele unabhängig vom Startspieler mit AB oder MTD(f) immer zu denselben Ergebnissen führen. Die Analyse zeigte jedoch, dass die Algorithmen nach einigen Zügen unterschiedliche Positionen als besten Zug wählten, wodurch sich das Endergebnis des Spiels änderte.

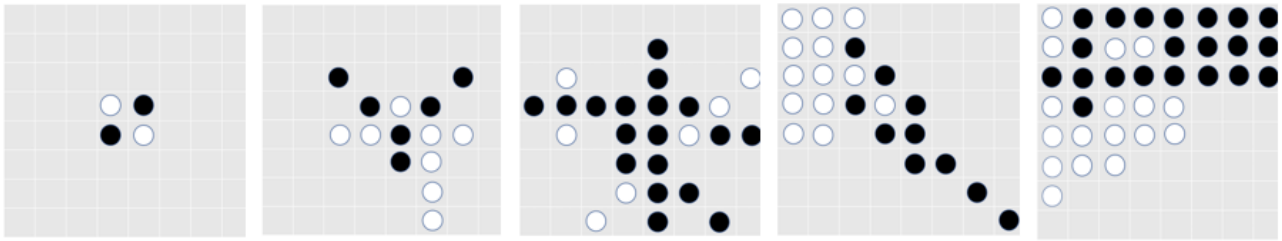


Abbildung 7.9.: Ausgewählte Testfälle aus der Abb. 6.4 zum Vergleich von Evaluationsmethoden. Dabei wurden die Testfälle *TF-01*, *TF-09*, *TF-12*, *TF-14* und *TF-20* für den Vergleich von Evaluationsmethoden ausgewählt.

	CoinParity	CornersCaptured	Mobility	Static	StaticMobility
CoinParity		4-6	5-5	2-8	3-7
CornersCaptured			6-4	3-7	2-8
Mobility				5-5	2-7
Static					3-7
StaticMobility					

StaticMobility	29
Static	23
CornersCaptured	17
Mobility	16
CoinParity	14
	99

Abbildung 7.10.: End-Ergebnisse von Vergleich von Evaluationsmethoden. In der Abbildung bedeutet *[a-b]* in einer Zelle, dass die Evaluationsmethode in Spalte 1 *a*-mal und die Evaluationsmethode in Spalte 2 *b*-mal gewonnen hat. Spiele, die unentschieden endeten, wurden bei der Berechnung ignoriert. Lediglich ein Spiel zwischen *Mobility* und *StaticMobility* endete unentschieden und wurde daher ignoriert (die gelb markierte Zelle). Aus diesem Grund wurden insgesamt nicht 100, sondern 99 Spiele in die Berechnung einbezogen. Die Mischmethode *StaticMobility* hat sich als die erfolgreichste Methode erwiesen, da sie mit 29 Siegen den ersten Platz belegte.

8. Fazit

In der vorliegenden Arbeit wurden verschiedene Suchalgorithmen und Evaluationsmethoden im Kontext eines selbst implementierten Othello-Spiels miteinander verglichen. Dabei wurden umfangreiche Tests durchgeführt und detaillierte Analysen präsentiert, um die Leistung und Effizienz der Algorithmen in verschiedenen Szenarien zu bewerten. Die Ergebnisse bieten wertvolle Erkenntnisse und können als Grundlage für zukünftige Forschungsarbeiten und Optimierungen in diesem Bereich dienen.

Die vergleichende Analyse der Suchalgorithmen zeigte, dass der MTD(f)-Algorithmus eine überlegene Alternative zum klassischen Alpha-Beta-Algorithmus ist. MTD(f) erreichte bessere Ergebnisse in Bezug auf die benötigte Zeit und die Anzahl der bearbeiteten Knoten. Besonders bei steigender Suchtiefe zeigte MTD(f) eine deutliche Verbesserung gegenüber Alpha-Beta. Diese Ergebnisse untermauern und bestätigen somit die Erkenntnisse aus [13] und [17].

Die Untersuchung der Evaluationsmethoden ergab, dass die Methode *StaticMobility*, eine Kombination aus statischen Gewichten und Mobilität, die besten Ergebnisse erzielte. Die vorliegenden Ergebnisse unterstützten die Erkenntnisse anderer wissenschaftlicher Arbeiten, siehe [3] und [15], zeigten jedoch auch, dass die Kombination verschiedener Evaluationsmethoden zu einer besseren Leistung führen kann.

Es wurden auch interessante Phänomene wie das *Tie-Breaking* bei der Entscheidung über den besten Zug in bestimmten Situationen beobachtet, die auf Unterschiede zwischen den Algorithmen hinweisen, wenn die Algorithmen bei gleichen Evaluationswerten entscheiden müssen.

Einige mögliche Verbesserungen könnten in zukünftigen Arbeiten angegangen werden, wie die Optimierung des Zobrist-Hashings oder die dynamische Berechnung von Evaluationswerten, siehe Kapitel 9. Diese Erkenntnisse und Vorschläge können als Ausgangspunkt für weitere Forschungen und Entwicklungen in diesem Bereich dienen.

9. Zukünftige Arbeiten

Es gibt einige Punkte, die sich negativ auf die Leistung und Laufzeit der Applikation auswirken, die bei zukünftigen Arbeiten verbessert werden könnten.

Wie bereits im Kapitel 5.2.1 zum Zobrist-Hashing erwähnt wurde, kann es bei einem Othello-Spiel vorkommen, dass in einem einzigen Zug eine große Anzahl von Steinen an den gegnerischen Spieler übergeht. Aus diesem Grund wurde bei der Berechnung des Zobrist-Hashings die Vorgehensweise gewählt, dass die Berechnung immer wieder von Anfang an erneut durchgeführt werden muss, was sich negativ auf die Laufzeit und Performance der Applikation auswirkt.

Normalerweise funktionieren Suchalgorithmen beim Suchen des besten Zuges folgendermaßen: Ein Zug wird ausgeführt, dann wird dieser Zug bewertet und anschließend wieder rückgängig gemacht. Aufgrund der oben genannten Problematik, dass in einem einzigen Zug eine große Anzahl von Steinen an den gegnerischen Spieler übergehen kann, wurde die Strategie verfolgt, dass während des Suchens des besten Zuges im Suchalgorithmus das Board-Objekt jedes Mal kopiert wird und die weiteren Operationen auf dieser Kopie durchgeführt werden. Diese Vorgehensweise erhöht jedoch den Speicherverbrauch und wirkt sich erneut negativ auf die Laufzeit und Performance aus, besonders bei der AB-Suche, wo ab Tiefe 8 eine enorme Anstieg von bearbeiteten Knoten beobachtet wurde, siehe die Abb. 7.2.

Des Weiteren wird bei den Evaluationsmethoden der Evaluationswert jedes Mal neu berechnet, was ebenfalls die Laufzeit negativ beeinflusst. Dieses Verhalten könnte in zukünftigen Arbeiten verbessert werden, indem die Implementierung eine dynamische Berechnung verwendet, anstatt bei jedem Zug eine neue Berechnung durchzuführen.

In der MTDF-Methode ist es üblich, feste Größe für die Transpositionstabellen mit Ersetzungsstrategien[1] zu verwenden. In der vorliegenden Arbeit wurde jedoch eine andere Methode angewandt, bei der die Transpositionstabellen ohne festgelegte Größe verwendet wurden. Dies führte jedoch zu einem erhöhten Speicherbedarf bei Tests ab einer Tiefe von 10, siehe die Abbildungen D.2, D.4 und D.5. Dieses Verhalten könnte bei zukünftigen Arbeiten verbessert werden.

[1] https://www.chessprogramming.org/Transposition_Table#cite_note-19 (abgerufen am 15.10.2023)

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

Abb.	Abbildung
Alg.	Algorithmus
Lst.	Listing
Tab.	Tabelle

A. Alle Ergebnisse: Wie schnell ermitteln die Algorithmen den besten Zug in jeder Tiefe

	Tiefe 5		
	AB	MTD(f)	BFM
TF-01	0,011	0,02	0,01
TF-02	0,022	0,032	0,014
TF-03	0,014	0,018	0,01
TF-04	0,018	0,027	0,13
TF-05	0,032	0,06	0,01
TF-06	0,053	0,031	0,013
TF-07	0,016	0,022	0,013
TF-08	0,043	0,026	0,013
TF-09	0,102	0,087	0,018
TF-10	0,116	0,072	0,014
TF-11	0,081	0,047	0,21
TF-12	0,076	0,056	0,008
TF-13	0,051	0,098	0,016
TF-14	0,022	0,041	0,001
TF-15	0,03	0,051	0,007
TF-16	0,077	0,06	0,017
TF-17	0.034	0,037	0,007
TF-18	0,017	0,025	0,016
TF-19	0,057	0,077	0,014
TF-20	0,008	0,012	0,012

	Tiefe 6		
	AB	MTD(f)	BFM
TF-01	0,02	0,019	0,021
TF-02	0,06	0,048	0,016
TF-03	0,018	0,018	0,011
TF-04	0,033	0,044	0,016
TF-05	0,046	0,053	0,012
TF-06	0,054	0,059	0,022
TF-07	0,056	0,091	0,01
TF-08	0,035	0,087	0,017
TF-09	0,121	0,161	0,029
TF-10	0,089	0,159	0,016
TF-11	0,126	0,178	0,019
TF-12	0,128	0,145	0,012
TF-13	0,385	0,194	0,017
TF-14	0,017	0,043	0,015
TF-15	0,103	0,114	0,011
TF-16	0,085	0,132	0,018
TF-17	0,063	0,071	0,015
TF-18	0,115	0,118	0,022
TF-19	0,085	0,079	0,018
TF-20	0,015	0,024	0,014

Abbildung A.1.: Wie schnell ermitteln die Algorithmen den besten Zug in jeder Tiefe (Tiefe 5-6). Zeitangaben sind in Sekunden berechnet.

	Tiefe 7		
	AB	MTD(f)	BFM
TF-01	0,039	0,041	0,015
TF-02	0,063	0,14	0,019
TF-03	0,048	0,033	0,015
TF-04	0,114	0,116	0,012
TF-05	0,32	0,342	0,016
TF-06	0,293	0,136	0,032
TF-07	0,069	0,087	0,014
TF-08	0,14	0,103	0,025
TF-09	0,494	0,679	0,026
TF-10	0,709	0,793	0,025
TF-11	0,234	0,231	0,019
TF-12	0,686	0,5	0,039
TF-13	0,315	0,449	0,032
TF-14	0,051	0,061	0,018
TF-15	0,193	0,197	0,017
TF-16	0,344	0,444	0,031
TF-17	0,132	0,131	0,035
TF-18	0,109	0,112	0,032
TF-19	0,217	0,293	0,029
TF-20	0,021	0,032	0,019

	Tiefe 8		
	AB	MTD(f)	BFM
TF-01	0,07	0,103	0,036
TF-02	0,417	0,212	0,029
TF-03	0,085	0,094	0,021
TF-04	0,182	0,226	0,041
TF-05	0,234	0,302	0,025
TF-06	0,292	0,599	0,065
TF-07	0,313	0,337	0,035
TF-08	0,234	0,987	0,037
TF-09	1,66	1,509	0,039
TF-10	1,278	1,782	0,029
TF-11	1,164	1,192	0,031
TF-12	2,543	1,874	0,027
TF-13	6,796	2,483	0,053
TF-14	0,093	0,104	0,029
TF-15	1,097	0,712	0,036
TF-16	1,13	1,387	0,031
TF-17	0,707	0,388	0,046
TF-18	1,087	0,749	0,051
TF-19	1,121	0,372	0,042
TF-20	0,074	0,056	0,036

Abbildung A.2.: Wie schnell ermitteln die Algorithmen den besten Zug in jeder Tiefe (Tiefe 7-8). Zeitangaben sind in Sekunden berechnet.

	Tiefe 9		
	AB	MTD(f)	BFM
TF-01	0,35	0,414	0,056
TF-02	0,424	0,674	0,048
TF-03	0,488	0,28	0,052
TF-04	0,893	0,87	0,117
TF-05	7,735	7,802	0,046
TF-06	3,472	1,489	0,16
TF-07	1,029	0,924	0,095
TF-08	1,608	0,995	0,093
TF-09	9,779	12,911	0,082
TF-10	15,704	23,58	0,067
TF-11	3,846	2,973	0,055
TF-12	8,526	10,866	0,09
TF-13	4,277	5,875	0,131
TF-14	0,342	0,386	0,053
TF-15	2,86	2,207	0,064
TF-16	6,66	7,544	0,083
TF-17	1,86	1,541	0,079
TF-18	1,711	0,898	0,184
TF-19	2,013	2,041	0,093
TF-20	0,195	0,191	0,137

	Tiefe 10		
	AB	MTD(f)	BFM
TF-01	0,095	0,913	0,099
TF-02	2,96	1,082	0,066
TF-03	1,1	0,984	0,06
TF-04	2,033	2,493	0,045
TF-05	3,2	3,603	0,059
TF-06	3,662	8,42	0,176
TF-07	4,281	4,92	0,085
TF-08	3,404	9,663	0,048
TF-09	139,421	135,365	0,13
TF-10	19,73	18,116	0,073
TF-11	15,573	10,886	0,063
TF-12	92,976	22,181	0,104
TF-13	210,803	53,489	0,136
TF-14	0,606	0,558	0,047
TF-15	14,214	6,155	0,049
TF-16	22,261	19,643	0,078
TF-17	13,018	3,812	0,077
TF-18	21,819	4,857	0,094
TF-19	26,735	2,494	0,091
TF-20	0,442	0,176	0,076

Abbildung A.3.: Wie schnell ermitteln die Algorithmen den besten Zug in jeder Tiefe (Tiefe 9-10). Zeitangaben sind in Sekunden berechnet.

B. Vergleich von Algorithmen ohne Zeitlimit

AB vs. BFM	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	26-38	38-26	31-33	53-11	20-44	55-9
TF-02	23-41	44-19	20-44	39-24	32-32	56-8
TF-03	29-35	48-16	25-39	46-18	29-35	54-10
TF-04	28-36	34-30	14-50	20-43	19-45	43-21
TF-05	26-38	29-35	22-42	54-10	18-46	47-17
TF-06	36-28	27-37	16-48	30-34	19-45	59-5
TF-07	28-36	33-31	9-55	41-23	15-49	48-16
TF-08	18-46	44-20	14-50	51-13	21-43	49-15
TF-09	22-42	44-20	20-44	48-16	26-38	46-18
TF-10	23-41	32-32	31-33	36-28	24-40	42-22
TF-11	21-43	40-24	12-52	43-21	16-48	47-17
TF-12	33-31	55-9	41-23	24-40	20-44	47-17
TF-13	26-38	51-12	36-28	34-29	24-40	16-48
TF-14	23-41	23-41	24-40	30-34	27-37	24-40
TF-15	29-35	48-16	45-19	54-9	40-24	43-21
TF-16	32-32	21-43	35-29	35-29	32-32	17-47
TF-17	17-47	48-16	31-33	20-44	45-19	48-16
TF-18	13-51	23-41	16-48	22-42	14-50	25-39
TF-19	25-38	38-26	34-30	25-39	28-36	38-26
TF-20	46-18	50-14	51-13	56-8	49-15	56-8

BFM vs. AB	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	8-56	22-42	51-13	17-47	32-32	15-49
TF-02	35-29	46-18	45-19	30-34	53-11	36-28
TF-03	39-25	50-14	39-25	29-35	33-31	29-35
TF-04	12-52	45-19	44-20	32-32	33-31	16-48
TF-05	28-36	28-36	45-19	43-21	44-20	15-49
TF-06	23-41	27-37	26-38	20-44	46-18	20-44
TF-07	29-35	36-28	18-46	33-31	41-23	31-33
TF-08	18-46	37-27	44-20	19-45	27-37	17-47
TF-09	21-43	28-36	47-17	22-42	50-14	18-46
TF-10	21-43	42-22	30-34	9-55	35-29	17-47
TF-11	27-37	36-28	41-23	11-53	24-40	9-55
TF-12	26-38	38-26	39-25	17-47	30-34	21-43
TF-13	14-50	29-35	27-37	25-39	41-23	23-41
TF-14	23-41	22-42	22-42	21-43	11-53	18-46
TF-15	20-44	40-24	30-34	26-38	34-30	27-37
TF-16	22-42	32-32	30-34	20-44	22-42	16-48
TF-17	24-40	39-25	31-33	17-47	42-22	16-48
TF-18	25-39	37-27	21-43	20-44	15-49	23-41
TF-19	29-35	36-28	26-38	24-40	25-39	28-36
TF-20	44-20	37-27	41-23	52-12	54-10	46-18

Abbildung B.1.: AB vs. BFM - Kein Zeitlimit. OOM steht für Out-Of-Memory.

AB vs. MTD(f)	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	59-5	40-24	19-45	33-31	22-42	25-39
TF-02	38-26	44-20	28-36	56-8	58-6	57-7
TF-03	45-19	56-8	51-13	17-47	17-47	OOM
TF-04	24-40	53-11	25-39	53-11	15-49	56-8
TF-05	8-56	41-23	33-31	23-41	11-53	55-9
TF-06	33-31	30-34	48-16	21-43	23-41	43-21
TF-07	20-44	33-31	54-10	50-14	57-7	OOM
TF-08	16-48	29-35	16-48	54-10	45-19	OOM
TF-09	29-35	49-15	39-25	10-54	27-37	49-15
TF-10	20-44	30-34	20-44	46-18	45-19	41-23
TF-11	26-38	39-25	17-47	22-42	13-51	18-46
TF-12	20-44	18-46	44-20	17-47	16-48	OOM
TF-13	29-35	41-23	20-44	28-36	18-46	39-25
TF-14	17-47	32-32	23-41	24-40	26-38	2-42
TF-15	39-25	28-36	52-12	36-28	49-15	30-34
TF-16	32-32	24-40	30-34	16-48	24-40	21-43
TF-17	48-16	29-35	34-30	25-39	47-17	30-34
TF-18	18-46	28-36	24-40	22-42	26-38	24-40
TF-19	34-30	23-41	26-38	33-31	34-30	26-38
TF-20	55-9	48-16	56-8	56-8	49-15	59-5

Abbildung B.2.: AB vs. MTD(f) - Kein Zeitlimit. OOM steht für Out-Of-Memory.

MTD(f) vs. BFM	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	16-48	37-27	26-38	23-41	42-22	45-19
TF-02	37-27	46-18	37-27	39-25	46-18	61-3
TF-03	11-53	46-18	37-27	53-10	44-20	33-31
TF-04	26-38	25-39	29-35	17-47	31-33	29-35
TF-05	32-32	22-42	23-41	47-17	19-45	51-13
TF-06	22-42	42-22	41-23	54-10	44-20	49-15
TF-07	37-27	30-34	14-49	47-17	16-48	54-10
TF-08	18-46	28-36	36-28	52-12	13-51	41-23
TF-09	23-41	52-12	25-39	45-19	14-50	OOM
TF-10	27-37	48-16	38-26	52-12	24-40	44-20
TF-11	27-37	45-19	13-51	55-9	19-45	37-27
TF-12	36-28	40-24	21-43	28-36	26-38	39-25
TF-13	48-16	31-33	38-26	45-19	39-25	30-34
TF-14	31-33	20-44	18-46	31-33	24-40	21-43
TF-15	45-19	29-35	37-27	52-12	32-32	47-17
TF-16	23-41	25-39	27-37	19-45	27-37	31-33
TF-17	41-23	27-37	42-22	22-42	38-26	47-17
TF-18	14-50	20-44	26-38	32-32	19-45	32-32
TF-19	30-34	27-37	28-36	36-28	20-44	24-40
TF-20	42-22	47-17	50-14	53-11	53-11	56-8

BFM vs. MTD(f)	Tiefe 5	Tiefe 6	Tiefe 7	Tiefe 8	Tiefe 9	Tiefe 10
TF-01	31-33	36-28	40-24	10-54	11-53	15-49
TF-02	55-9	48-16	41-23	47-17	44-20	33-31
TF-03	39-25	19-45	34-30	22-42	43-21	44-20
TF-04	40-24	30-34	43-21	16-48	22-42	21-43
TF-05	45-19	34-30	17-47	21-43	31-33	OOM
TF-06	37-27	14-50	34-30	13-51	42-22	OOM
TF-07	33-31	13-51	34-30	43-21	29-35	25-39
TF-08	22-42	26-38	42-22	8-56	21-43	33-31
TF-09	50-14	18-46	31-33	24-40	30-34	OOM
TF-10	31-33	18-46	26-38	8-56	37-27	45-19
TF-11	34-30	27-37	28-36	20-44	39-25	21-43
TF-12	33-30	14-50	5-59	20-44	38-26	16-48
TF-13	49-15	20-44	22-42	12-52	42-21	20-44
TF-14	15-49	20-44	26-38	19-45	25-39	2-42
TF-15	22-42	23-41	48-16	24-40	41-23	19-45
TF-16	15-49	16-48	25-39	28-36	25-39	15-49
TF-17	26-38	42-22	44-20	20-44	40-24	33-31
TF-18	14-50	27-37	26-38	23-41	21-43	14-50
TF-19	26-38	18-46	24-40	29-35	22-42	27-37
TF-20	50-14	44-20	41-23	45-19	53-11	49-15

Abbildung B.3.: MTD(f) vs. BFM - Kein Zeitlimit. OOM steht für Out-Of-Memory.

C. Vergleich von Evaluationsmethoden

	CoinParity vs. CornersCaptured	CornersCaptured vs. CoinParity	CoinParity	CornersCaptured
TF-01	37-27	57-7	44	84
TF-09	11-53	50-13	24	103
TF-12	18-46	10-54	72	56
TF-14	8-56	14-49	57	70
TF-20	56-2	51-13	69	53
			266	366

	CoinParity vs. Mobility	Mobility vs. CoinParity	CoinParity	Mobility
TF-01	42-0	0-30	72	0
TF-09	7-57	46-18	25	140
TF-12	11-53	24-40	51	77
TF-14	16-48	5-59	75	53
TF-20	57-7	46-18	75	53
			298	323

	CoinParity vs. Static	Static vs. CoinParity	CoinParity	Static
TF-01	23-41	44-20	43	85
TF-09	19-45	38-26	45	83
TF-12	14-50	46-18	32	96
TF-14	23-41	9-55	78	50
TF-20	56-2	48-16	72	50
			270	364

	CoinParity vs. StaticMobility	StaticMobility vs. CoinParity	CoinParity	StaticMobility
TF-01	23-0	48-16	39	48
TF-09	24-40	63-0	24	103
TF-12	20-44	37-27	47	81
TF-14	17-47	22-42	59	69
TF-20	56-2	53-11	67	55
			236	356

	CornersCaptured vs. Mobility	Mobility vs. CornersCaptured	CornersCaptu	Mobility
TF-01	52-12	42-22	74	54
TF-09	21-43	10-54	75	53
TF-12	38-26	25-39	77	51
TF-14	23-41	25-39	62	66
TF-20	46-18	50-14	60	68
			348	292

Abbildung C.1.: Vergleich von Evaluationsmethoden - Part 1

	CornersCaptured vs. Static	Static vs. CornersCaptured	CornersCaptu	Static
TF-01	19-45	33-0	19	78
TF-09	36-28	39-25	61	67
TF-12	21-43	39-25	46	82
TF-14	20-44	20-44	64	64
TF-20	37-27	53-11	48	80
			238	371

	CornersCaptured vs. StaticMobili	StaticMobility vs. CornersCaptured	CornersCaptu	StaticMobility
TF-01	13-51	55-9	22	106
TF-09	27-37	59-5	32	96
TF-12	15-49	34-30	45	83
TF-14	1-61	25-39	40	86
TF-20	40-24	51-13	53	75
			192	446

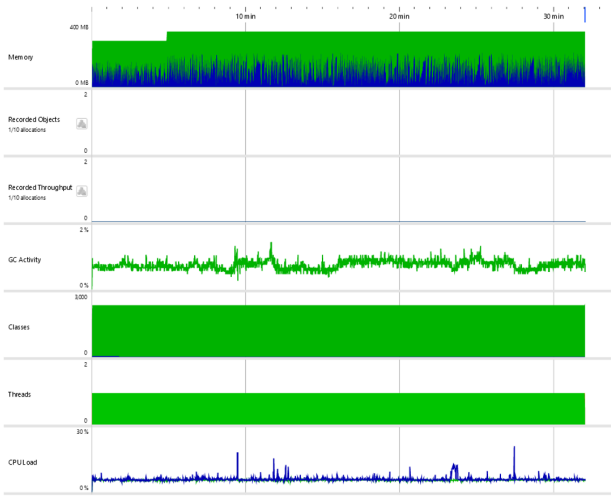
	Mobility vs. Static	Static vs. Mobility	Mobility	Static
TF-01	20-44	28-36	56	72
TF-09	18-46	30-34	52	76
TF-12	26-38	31-33	59	69
TF-14	12-52	22-42	54	74
TF-20	37-27	52-12	49	79
			270	370

	Mobility vs. StaticMobility	StaticMobility vs. Mobility	Mobility	StaticMobility
TF-01	18-46	43-21	39	89
TF-09	8-56	32-32	40	88
TF-12	29-35	38-26	55	73
TF-14	10-54	26-38	48	80
TF-20	58-3	50-14	72	53
			254	383

	Static vs. StaticMobility	StaticMobility vs. Static	Static	StaticMobility
TF-01	24-40	45-19	43	85
TF-09	22-42	45-19	41	87
TF-12	12-52	30-34	46	82
TF-14	2-60	21-43	45	81
TF-20	54-10	49-15	69	59
			244	394

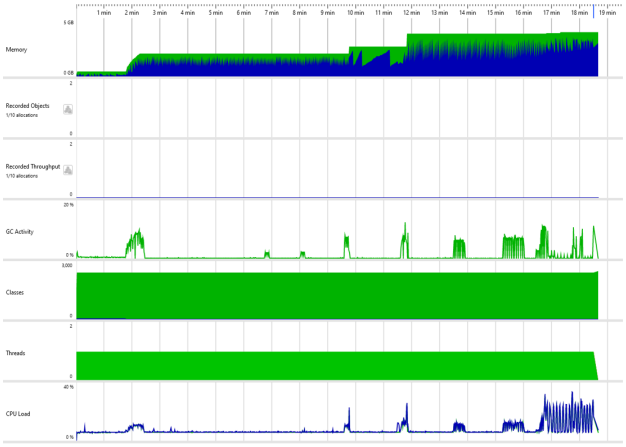
Abbildung C.2.: Vergleich von Evaluationsmethoden - Part 2

D. Vergleich von Algorithmen per JProfiler



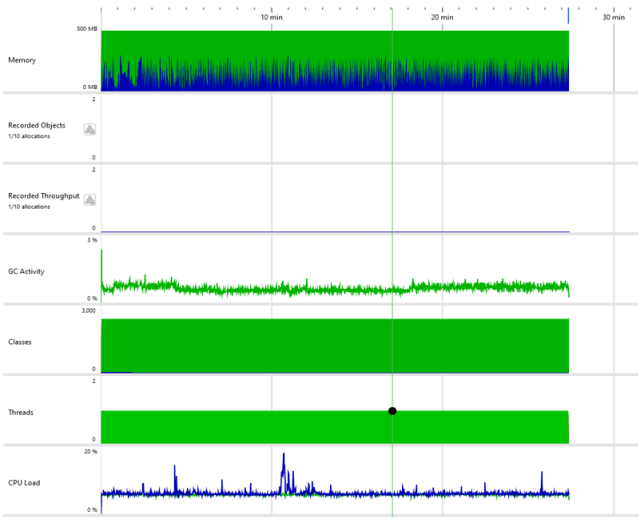
Name	Instance Count	Size
berlin.tu [15 classes]	1,086,859 (99 %)	50,094 kB
berlin.tu.Cell[ ]	565,562 (52 %)	27,146 kB
berlin.tu.Direction[ ]	261,952 (24 %)	12,573 kB
berlin.tu.Position	117,933 (10 %)	4,717 kB
berlin.tu.Cell[ ][ ]	70,695 (6 %)	3,393 kB
berlin.tu.Board	70,695 (6 %)	2,262 kB
berlin.tu.Direction	8 (0 %)	256 bytes
berlin.tu.Game	1 (0 %)	80 bytes
berlin.tu.Cell	3 (0 %)	72 bytes
berlin.tu.MoveState	2 (0 %)	48 bytes
berlin.tu.Player	2 (0 %)	48 bytes
berlin.tu.State	2 (0 %)	48 bytes
berlin.tu.AllGamesTest	1 (0 %)	40 bytes
berlin.tu.MoveState[ ]	1 (0 %)	24 bytes
berlin.tu.Player[ ]	1 (0 %)	24 bytes
berlin.tu.State[ ]	1 (0 %)	24 bytes
berlin.tu.algorithms [1 classes]	2 (0 %)	96 bytes
berlin.tu.algorithms.AlphaBeta	2 (0 %)	96 bytes
berlin.tu.evaluations [1 classes]	2 (0 %)	32 bytes
berlin.tu.evaluations.StaticEvaluation	2 (0 %)	32 bytes
Total from 3 rows:	**1,086,863 (100 %)**	**50,094 kB**

Abbildung D.1.: Profilerergebnisse AB vs. AB für den Testfall *TF-09* aus der Abb. 6.4. Das Spiel wurde nur für die Tiefe 10 gespielt.



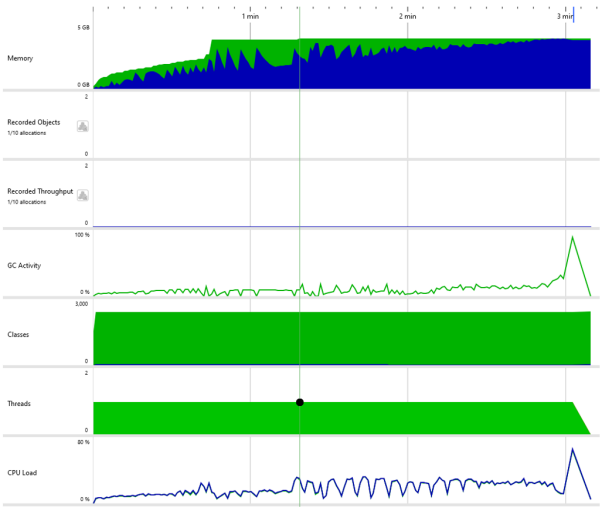
Name	Instance Count	Size
berlin.tu.zobrist [2 classes]	28,981,241 (71 %)	695 MB
berlin.tu.zobrist.TranspositionTableEntry	28,981,240 (71 %)	695 MB
berlin.tu.zobrist.ZobristHash	1 (0 %)	32 bytes
berlin.tu [15 classes]	11,394,530 (28 %)	520 MB
berlin.tu.Cell[ ]	4,414,914 (10 %)	211 MB
berlin.tu.Direction[ ]	3,660,345 (9 %)	175 MB
berlin.tu.Position	2,215,521 (5 %)	88,620 kB
berlin.tu.Cell[ ][ ]	551,864 (1 %)	26,489 kB
berlin.tu.Board	551,864 (1 %)	17,659 kB
berlin.tu.Direction	8 (0 %)	256 bytes
berlin.tu.Game	1 (0 %)	80 bytes
berlin.tu.Cell	3 (0 %)	72 bytes
berlin.tu.MoveState	2 (0 %)	48 bytes
berlin.tu.Player	2 (0 %)	48 bytes
berlin.tu.State	2 (0 %)	48 bytes
berlin.tu.AllGamesTest	1 (0 %)	40 bytes
berlin.tu.MoveState[ ]	1 (0 %)	24 bytes
berlin.tu.Player[ ]	1 (0 %)	24 bytes
berlin.tu.State[ ]	1 (0 %)	24 bytes
berlin.tu.algorithms [2 classes]	2 (0 %)	96 bytes
berlin.tu.algorithms.AlphaBeta	1 (0 %)	48 bytes
berlin.tu.algorithms.MTDF	1 (0 %)	48 bytes
berlin.tu.evaluations [1 classes]	2 (0 %)	32 bytes
berlin.tu.evaluations.StaticEvaluation	2 (0 %)	32 bytes
Total from 4 rows:	**40,375,775 (100 %)**	**1,215 MB**

Abbildung D.2.: Profilerergebnisse AB vs. MTD(f) für den Testfall *TF-09* aus der Abb. 6.4. Das Spiel wurde nur für die Tiefe 10 gespielt.



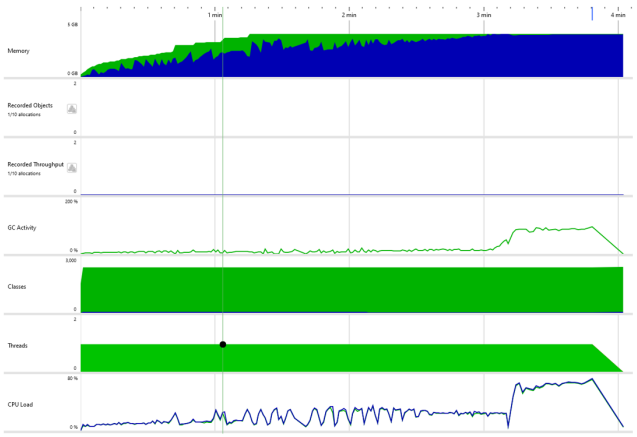
Name	Instance Count	Size
berlin.tu [16 classes]	3,465,701 (99 %)	158 MB
berlin.tu.Cell[ ]	1,425,626 (41 %)	68,430 kB
berlin.tu.Direction[ ]	1,083,283 (31 %)	51,997 kB
berlin.tu.Position	569,066 (16 %)	22,762 kB
berlin.tu.Cell[ ][ ]	178,203 (5 %)	8,553 kB
berlin.tu.Board	178,203 (5 %)	5,702 kB
berlin.tu.MoveEvaluation	31,298 (0 %)	751 kB
berlin.tu.Direction	8 (0 %)	256 bytes
berlin.tu.Game	1 (0 %)	80 bytes
berlin.tu.Cell	3 (0 %)	72 bytes
berlin.tu.MoveState	2 (0 %)	48 bytes
berlin.tu.Player	2 (0 %)	48 bytes
berlin.tu.State	2 (0 %)	48 bytes
berlin.tu.AllGamesTest	1 (0 %)	40 bytes
berlin.tu.MoveState[ ]	1 (0 %)	24 bytes
berlin.tu.Player[ ]	1 (0 %)	24 bytes
berlin.tu.State[ ]	1 (0 %)	24 bytes
berlin.tu.algorithms [2 classes]	2 (0 %)	96 bytes
berlin.tu.algorithms.AlphaBeta	1 (0 %)	48 bytes
berlin.tu.algorithms.BestFirstMinimax	1 (0 %)	48 bytes
berlin.tu.evaluations [1 classes]	2 (0 %)	32 bytes
berlin.tu.evaluations.StaticEvaluation	2 (0 %)	32 bytes
Total from 3 rows:	**3,465,705 (100 %)**	**158 MB**

Abbildung D.3.: Profilerergebnisse AB vs. BFM für den Testfall *TF-09* aus der Abb. 6.4. Das Spiel wurde nur für die Tiefe 10 gespielt.



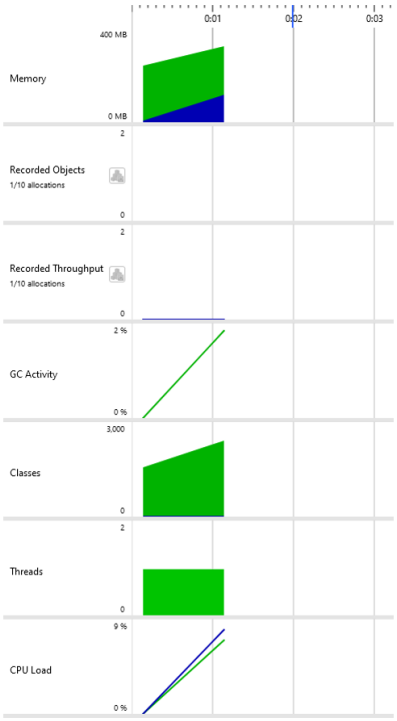
Name	Instance Count	Size
berlin.tu.zobrist [2 classes]	48,628,898 (99 %)	1,167 MB
berlin.tu.zobrist.TranspositionTableEntry	48,628,896 (99 %)	1,167 MB
berlin.tu.zobrist.ZobristHash	2 (0 %)	64 bytes
berlin.tu [15 classes]	372 (0 %)	16,064 bytes
berlin.tu.Cell[ ]	178 (0 %)	8,512 bytes
berlin.tu.Position	127 (0 %)	5,080 bytes
berlin.tu.Cell[ ][ ]	22 (0 %)	1,056 bytes
berlin.tu.Board	22 (0 %)	704 bytes
berlin.tu.Direction	8 (0 %)	256 bytes
berlin.tu.Game	1 (0 %)	80 bytes
berlin.tu.Cell	3 (0 %)	72 bytes
berlin.tu.Direction[ ]	1 (0 %)	48 bytes
berlin.tu.MoveState	2 (0 %)	48 bytes
berlin.tu.Player	2 (0 %)	48 bytes
berlin.tu.State	2 (0 %)	48 bytes
berlin.tu.AllGamesTest	1 (0 %)	40 bytes
berlin.tu.MoveState[ ]	1 (0 %)	24 bytes
berlin.tu.Player[ ]	1 (0 %)	24 bytes
berlin.tu.State[ ]	1 (0 %)	24 bytes
berlin.tu.algorithms [1 classes]	2 (0 %)	96 bytes
berlin.tu.algorithms.MTDF	2 (0 %)	96 bytes
berlin.tu.evaluations [1 classes]	2 (0 %)	32 bytes
berlin.tu.evaluations.StaticEvaluation	2 (0 %)	32 bytes
Total from 4 rows:	**48,629,274 (100 %)**	**1,167 MB**

Abbildung D.4.: Profilerergebnisse MTD(f) vs. MTD(f) für den Testfall *TF-09* aus der Abb. 6.4. Das Spiel wurde nur für die Tiefe 10 gespielt. Dieses Spiel endete mit einer *Out Of Memory (OOM)* Exception.



Name	Instance Count	Size
berlin.tu.zobrist [2 classes]	50,331,650 (99 %)	1,207 MB
berlin.tu.zobrist.TranspositionTableEntry	50,331,649 (99 %)	1,207 MB
berlin.tu.zobrist.ZobristHash	1 (0 %)	32 bytes
berlin.tu [15 classes]	368 (0 %)	15,840 bytes
berlin.tu.Cell[ ]	170 (0 %)	8,128 bytes
berlin.tu.Position	133 (0 %)	5,320 bytes
berlin.tu.Cell[ ][ ]	21 (0 %)	1,008 bytes
berlin.tu.Board	21 (0 %)	672 bytes
berlin.tu.Direction	8 (0 %)	256 bytes
berlin.tu.Game	1 (0 %)	80 bytes
berlin.tu.Cell	3 (0 %)	72 bytes
berlin.tu.Direction[ ]	1 (0 %)	48 bytes
berlin.tu.MoveState	2 (0 %)	48 bytes
berlin.tu.Player	2 (0 %)	48 bytes
berlin.tu.State	2 (0 %)	48 bytes
berlin.tu.AllGamesTest	1 (0 %)	40 bytes
berlin.tu.MoveState[ ]	1 (0 %)	24 bytes
berlin.tu.Player[ ]	1 (0 %)	24 bytes
berlin.tu.State[ ]	1 (0 %)	24 bytes
berlin.tu.algorithms [2 classes]	2 (0 %)	96 bytes
berlin.tu.algorithms.BestFirstMinimax	1 (0 %)	48 bytes
berlin.tu.algorithms.MTDF	1 (0 %)	48 bytes
berlin.tu.evaluations [1 classes]	2 (0 %)	32 bytes
berlin.tu.evaluations.StaticEvaluation	2 (0 %)	32 bytes
Total from 4 rows:	**50,332,022 (100 %)**	**1,207 MB**

Abbildung D.5.: Profilerergebnisse MTD(f) vs. BFM für den Testfall *TF-09* aus der Abb. 6.4. Das Spiel wurde nur für die Tiefe 10 gespielt. Dieses Spiel endete mit einer *Out Of Memory (OOM)* Exception.



Name	Instance Count	Size
berlin.tu [16 classes]	587,150 (99 %)	25,202 kB
berlin.tu.Position	272,194 (46 %)	10,887 kB
berlin.tu.Cell[ ]	158,938 (27 %)	7,628 kB
berlin.tu.Direction[ ]	96,051 (16 %)	4,610 kB
berlin.tu.Cell[ ][ ]	19,867 (3 %)	953 kB
berlin.tu.Board	19,867 (3 %)	635 kB
berlin.tu.MoveEvaluation	20,211 (3 %)	485 kB
berlin.tu.Direction	8 (0 %)	256 bytes
berlin.tu.Game	1 (0 %)	80 bytes
berlin.tu.Cell	3 (0 %)	72 bytes
berlin.tu.MoveState	2 (0 %)	48 bytes
berlin.tu.Player	2 (0 %)	48 bytes
berlin.tu.State	2 (0 %)	48 bytes
berlin.tu.AllGamesTest	1 (0 %)	40 bytes
berlin.tu.MoveState[ ]	1 (0 %)	24 bytes
berlin.tu.Player[ ]	1 (0 %)	24 bytes
berlin.tu.State[ ]	1 (0 %)	24 bytes
berlin.tu.algorithms [1 classes]	2 (0 %)	96 bytes
berlin.tu.algorithms.BestFirstMinimax	2 (0 %)	96 bytes
berlin.tu.evaluations [1 classes]	2 (0 %)	32 bytes
berlin.tu.evaluations.StaticEvaluation	2 (0 %)	32 bytes
Total from 3 rows:	**587,154 (100 %)**	**25,202 kB**

Abbildung D.6.: Profilerergebnisse BFM vs. BFM für den Testfall *TF-09* aus der Abb. 6.4. Das Spiel wurde nur für die Tiefe 10 gespielt.

Literaturverzeichnis

[1] Der alpha-beta-algorithmus. http://www.inf.fu-berlin.de/lehre/SS17/PSThInf/notes/03_alphabeta.pdf. (abgerufen am 09.10.2023).

[2] Strategy guide for reversi and reversed reversi. https://www.samsoft.org.uk/reversi/strategy.htm. (abgerufen am 28.08.2023).

[3] Alec Barber, Warren Pretorius, Uzair Qureshi, and Dhruv Kumar. An analysis of othello ai strategies.

[4] Jeroen WT Carolus. Alpha-beta with sibling prediction pruning in chess. *Amsterdam: University of Amsterdam*, 2006.

[5] Jack Chen. Applications of artificial intelligence and machine learning in othello. *Computer Systems Lab at Thomas Jefferson High School of Science & Technology*, 2010.

[6] Jorge Hernandez, Karen Daza, and Hector Florez. Alpha-beta vs scout algorithms for the othello game. In *CEUR Workshops Proceedings*, volume 2846, pages 65–79, 2019.

[7] Shigeki Iwata and Takumi Kasai. The othello game on an n × n board is pspace-complete. *Theoretical Computer Science*, 123(2):329–340, 1994.

[8] Richard E Korf and David Maxwell Chickering. Best-first minimax search. *Artificial intelligence*, 84(1-2):299–337, 1996.

[9] Michael Korman. Playing othello with artificial intelligence. 2003.

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

[11] Paweł Liskowski, Wojciech Jaśkowski, and Krzysztof Krawiec. Learning to play othello with deep neural networks. *IEEE Transactions on Games*, 10:354–364, 2017.

[12] Kevin Lu. The game theory of reversi. 2014.

[13] Aske Plaat. A minimax algorithm faster than negascout. *Rotterdam: Erasmus University*, 1997.

[14] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first and depth-first minimax search in practice. *arXiv preprint arXiv:1505.01603*, 2015.

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

[16] Brian Sumali, Ivan Michael Siregar, and Rosalina Rosalina. Implementation of minimax with alpha-beta pruning as computer player in congklak. *Jurnal Teknik Informatika dan Sistem Informasi*, 2(2), 2016.

[17] Lukas Tommy, Mardi Hardjianto, and Nazori Agani. The analysis of alpha beta pruning and mtd(f) algorithm to determine the best algorithm to be implemented at connect four prototype. *IOP Conference Series: Materials Science and Engineering*, 190(1):012044, apr 2017.

[18] Albert L Zobrist. A new hashing method with application for game playing. *ICGA Journal*, 13(2):69–73, 1990.