Systematischer Vergleich von Alpha-Beta, PVS, MTD(f) und MCTS Algorithmen im Kontext des Spiels Amazons

Bachelorarbeit

Timm Marvin Spangenberg
402402

15. November 2024

Gutachter: Prof. Dr. Benjamin Blankertz
Dr.-Ing Stefan Fricke

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

Kurzfassung

Die Forschung im Bereich der Spieltheorie beschäftigt sich mit der Entwicklung effizienter Suchalgorithmen zur Lösung komplexer Entscheidungsprobleme in Spielen. Das strategisch anspruchsvolle Spiel *Amazons* bietet ein ideales Testfeld für den Vergleich verschiedener Suchalgorithmen. Aufgrund seiner hohen Komplexität und des großen Suchraums, stellt das Brettspiel eine sehr große Herausforderung für die Algorithmen dar.

Die vorliegende Arbeit hat das Ziel, einen systematischen Vergleich der Algorithmen *Alpha-Beta*, *Principal Variation Search (PVS)*, *MTD(f)* und *Monte Carlo Tree Search (MCTS)* im Kontext des Spiels *Amazons* durchzuführen. Dabei werden die Algorithmen auf ihre Leistungsfähigkeit und Effizienz unter verschiedenen Bedingungen untersucht. Die Beantwortung der Forschungsfrage umfasst die Implementierung und Anpassung der Algorithmen für das Spiel Amazons, gefolgt von einer Reihe von Tests, in denen die Algorithmen gegeneinander antreten.

Die Ergebnisse verdeutlichen die spezifischen Stärken und Schwächen der Algorithmen im direkten Vergleich. Alpha-Beta hebt sich insgesamt deutlich in der Spielstärke von den anderen Ansätzen ab. PVS spielt seine Stärken insbesondere in der frühen und zusammen mit MTD(f), in der mittleren Spielphase aus, während MCTS seine Fähigkeiten vor allem in der Endspielphase zeigt. Dennoch bleibt MCTS hinsichtlich der Spielstärke deutlich hinter Alpha-Beta und PVS zurück.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das Spiel *Amazons* ist ein strategisches Brettspiel, das aufgrund seiner hohen Komplexität und der Vielzahl möglicher Züge pro Spielzug eine besondere Herausforderung für die Entwicklung von Suchalgorithmen darstellt. Ähnlich wie bei anderen baumbasierten Spielen besteht die zentrale Aufgabe darin, den besten Zug in einem großen Suchraum effizient zu finden. Da es im Durchschnitt mehr als 1000 legale Züge pro Position gibt [6], erfordert die Lösung dieses Problems optimierte Algorithmen, die mit dieser enormen Verzweigung effizient umgehen können.

Um optimale oder zumindest gute Züge in solch einem komplexen Spiel zu berechnen, werden Suchalgorithmen eingesetzt. Es gibt verschiedene Ansätze, die auf unterschiedlichen Techniken basieren, wie z.B. *Alpha-Beta-Pruning*, *Principal Variation Search* (PVS), *MTD(f)* und *Monte Carlo Tree Search* (MCTS). Jeder dieser Algorithmen hat seine Stärken und Schwächen in Bezug auf Rechenzeit, Genauigkeit und Effizienz bei großen Spielbäumen.

1.1 Zielsetzung und Relevanz

Das Ziel ist es, dass die Leistungsfähigkeit der verschiedenen Suchalgorithmen: *Alpha-Beta*, *Principal Variation Search (PVS)*, *MTD(f)* und *Monte Carlo Tree Search (MCTS)* für das Brettspiel Amazons untersucht und verglichen werden. Dabei gilt es systematisch herauszufinden, welcher dieser Algorithmen unter welchen Bedingungen die besten Ergebnisse in unterschiedlichen Aspekten erzielt und welche Stärken und Schwächen die Algorithmen im direkten Vergleich aufweisen. Durch die Analyse der Ergebnisse sollen dann fundierte Aussagen über die Eignung und Effizienz dieser Algorithmen für komplexe strategische Spiele wie *Amazons* getroffen werden.

Die Relevanz dieses Vergleichs ergibt sich aus der hohen Komplexität und den besonderen Herausforderungen, die das Spiel *Amazons* für Suchalgorithmen darstellt. Aufgrund der dynamischen und sich ständig verändernden Spielumgebung sowie der großen Anzahl möglicher Züge pro Spielposition, ist es für klassische Suchalgorithmen eine Herausforderung, optimale Züge effizient zu berechnen. Ein tieferes Verständnis der Leistungsfähigkeit dieser Algorithmen kann nicht nur zur Entwicklung stärkerer Spielprogramme beitragen, sondern auch wertvolle Erkenntnisse für andere Bereiche der Künstlichen Intelligenz liefern, in denen ähnliche Herausforderungen bestehen. Die Ergebnisse dieser Arbeit könnten somit nicht nur für die Forschung im Bereich der Spieltheorie und Künstlichen Intelligenz relevant sein, sondern auch für praktische Anwendungen, bei denen effiziente Entscheidungsfindung in komplexen und dynamischen Umgebungen erforderlich ist.

1.2 Ähnliche Arbeiten

Die Entwicklung und Analyse von Algorithmen zur Lösung des Brettspiels hat in der Forschung zahlreiche Ansätze hervorgebracht, darunter Monte Carlo-Methoden, Alpha-Beta-Pruning, MTD(f) und Principal Variation Search (PVS). In früheren Arbeiten wurde untersucht, wie diese Algorithmen sich in ihrer Effizienz und Effektivität im Kontext des Spiels Amazons verhalten und wie sie durch verschiedene Heuristiken und Optimierungstechniken verbessert werden können.

Kloetzer et al. untersuchten in einer vergleichenden Analyse die Effektivität verschiedener Lösungsalgorithmen für die Endspielphase. Sie zeigten, dass Monte Carlo-Methoden aufgrund des hohen Verzweigungsfaktors eine geeignete Alternative zu klassischen Minimax-basierten Methoden darstellen und in Kombination mit Baumstruktursuchen starke Ergebnisse erzielen können [2]. Auch Lorentz kombinierte Monte Carlo-Ansätze mit UCT-Algorithmen, was eine verbesserte Spielstärke des Programms „InvaderMC“ bewirkte, indem hybride Ansätze (Kombination aus Monte Carlo Tree Search und Alpha-Beta) genutzt wurden [6]. Jianning et al. analysierten den Upper Confidence bounds applied to Trees (UCT)-Algorithmus und zeigten dessen Vorteile gegenüber traditionellen Alpha-Beta-Verfahren in Bezug auf die Effizienz bei der Reduzierung des Suchraums [13].

Kato et al. führten eine vergleichende Studie zwischen Monte Carlo Tree Search und Alpha-Beta-Pruning durch. Sie fanden heraus, dass Alpha-Beta-Pruning durch eine bessere Auswahl an Bewertungsfunktionen gestärkt werden kann, während eine bloße Erhöhung der Simulationen bei MCTS keine signifikante Verbesserung der Spielstärke bringt [14]. Bouzy, Iida und Kloetzer stellten eine modifizierte Version von MCTS vor, die speziell für Amazons entwickelt wurde. Sie untersuchten, wie die Kombination von MCTS mit einer Bewertungsfunktion die Spielstärke verbessert, insbesondere durch die Nutzung von *Tree Search*-Erweiterungen und einer Optimierung der Zufallssimulationen (*Playouts*). Ihre Forschung unterstreicht die Bedeutung der Playout-Optimierung in MCTS, um bessere Ergebnisse zu erzielen [1].

Weitere Untersuchungen zur Leistung von Suchalgorithmen für Amazons führten Qiu et al. durch. Sie untersuchten die Anwendung von PVS, MTD(f) und MTD(bi) und zeigten, dass die PVS-Methoden effizienter sind, wenn sie mit Heuristiken wie der Sortierung und Transpositionstabellen kombiniert werden [7]. Ebenfalls hat J. Kloetzer et al. die Anwendung des *MTD(f)*-Algorithmus auf das Spiel untersucht. Die Ergebnisse zeigen, dass *MTD(f)* besonders in Endspiel-Szenarien effektiv ist, jedoch für frühe Spielphasen mit hoher Unsicherheit weniger geeignet scheint [2].

Die Optimierung der PVS-Algorithmen im Rahmen von Parallelisierungsstrategien wurde ebenfalls untersucht. Guo et al. zeigten, dass durch die Einbeziehung von Transpositionstabellen und parallelen Berechnungen die Effizienz und die Spielstärke des PVS-Algorithmus erheblich gesteigert werden konnten [11].

1.3 Struktur der Arbeit

Die vorliegende Arbeit ist in sieben Kapitel unterteilt und lässt sich wie folgt gliedern:

- **Kapitel 1 – Einleitung**: Dieses Kapitel führt in die Problematik und die Zielsetzung der Arbeit ein. Hier werden die Relevanz des Themas und die Motivation für den systematischen Vergleich von Suchalgorithmen dargelegt.
- **Kapitel 2 – Amazons**: In diesem Kapitel werden die Regeln und die strategischen Aspekte des Brettspiels vorgestellt. Dabei werden sowohl die Spielregeln als auch wesentliche strategische Überlegungen und Eigenschaften des Spiels erläutert.
- **Kapitel 3 – Algorithmen zur Spielbaum-Suche**: Dieses Kapitel widmet sich der detaillierten Beschreibung der vier zu vergleichenden Algorithmen: Alpha-Beta, Principal Variation Search (PVS), MTD(f) und Monte Carlo Tree Search (MCTS). Jeder Algorithmus wird mit seinen Funktionsprinzipien und Optimierungen vorgestellt.
- **Kapitel 4 – Durchführung der Tests**: Hier wird der Testaufbau zur Untersuchung der Algorithmen beschrieben. Es werden die Testumgebung, Testaufbau, die Testmetriken sowie der Ablauf der Testreihen dargelegt und am Ende des Kapitels Herausforderungen und mögliche Optimierungen erläutert.
- **Kapitel 5 – Ergebnisse**: In diesem Kapitel werden die Ergebnisse der durchgeführten Tests präsentiert. Die Siegquoten, die Anzahl der Iterationen pro Zug, die Laufzeiten der Algorithmen und weitere Testergebnisse werden gegenübergestellt.
- **Kapitel 6 – Diskussion**: In diesem Kapitel werden die Ergebnisse analysiert und im Hinblick auf die Stärken und Schwächen der verschiedenen Algorithmen diskutiert und interpretiert. Optimierungspotenziale und mögliche Verbesserungen der Algorithmen werden aufgezeigt.
- **Kapitel 7 – Fazit**: Abschließend wird eine Zusammenfassung der Arbeit gegeben und es werden die wichtigsten Erkenntnisse zusammengefasst.

2 Amazons

Amazons oder auch *Das Spiel der Amazonen* ist ein strategisches Brettspiel für zwei Spieler, das von dem Argentinier Walter Zamkauskas im Jahr 1988 erfunden wurde [6]. Das Spiel kombiniert Elemente aus den bekannten Brettspielen Schach und Go.

2.1 Spielregeln

Amazons wird typischerweise auf einem $10x10$ Schachbrett gespielt, dabei hat jeder Spieler auf seiner Spielfeldseite vier Amazonen (oder auch Damen genannt), die am Anfang des Spiels wie in Abbildung 2.1 platziert werden.

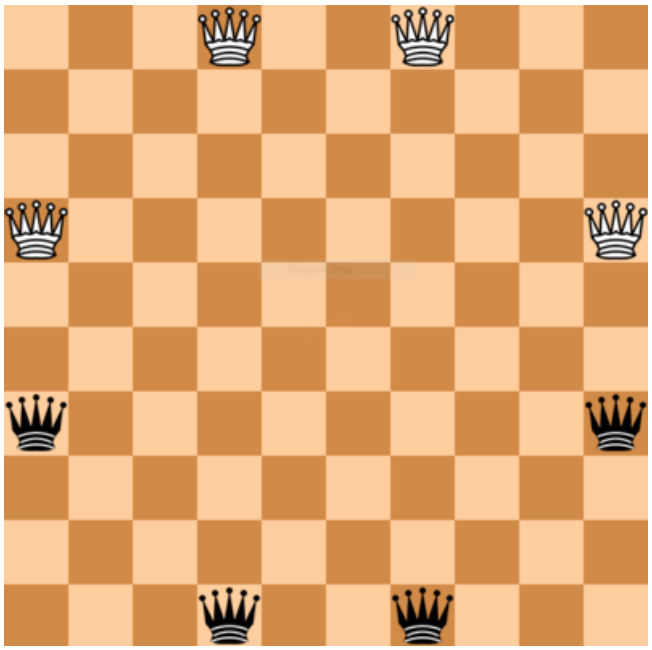

Abbildung 2.1: Amazons Startposition [12]

Der Spieler 1 beginnt das Spiel mit den weißen Spielfiguren, gefolgt von Spieler 2, der die schwarzen Spielfiguren führt. Die Züge werden abwechselnd gespielt und die Spieler haben jeweils eine **Zugpflicht**.

Ein Zug eines Spielers besteht immer aus **zwei Teilen** und muss auch in dieser Reihenfolge ausgeführt werden:

1. **Amazone bewegen**: Die Amazone muss am Anfang jeden Zuges bewegt werden. Die Bewegung gleicht der einer Dame von Schach. Dabei kann die Amazone des Spielers über mehrere Felder in orthogonaler oder diagonaler Richtung gezogen werden. Dabei gilt es jedoch zu beachten, dass die Amazone nur auf ein leeres Feld gezogen werden darf und dabei auch nur ohne das Überqueren von anderen Amazonen oder Pfeilen. In Abbildung 2.2 werden zwei mögliche Zugbewegungen der weißen Amazone illustriert. Dagegen wird in Abbildung 2.3 gezeigt, welche zwei Züge für Schwarz illegal sind, da sie eine andere Amazone oder Pfeil überqueren würden.

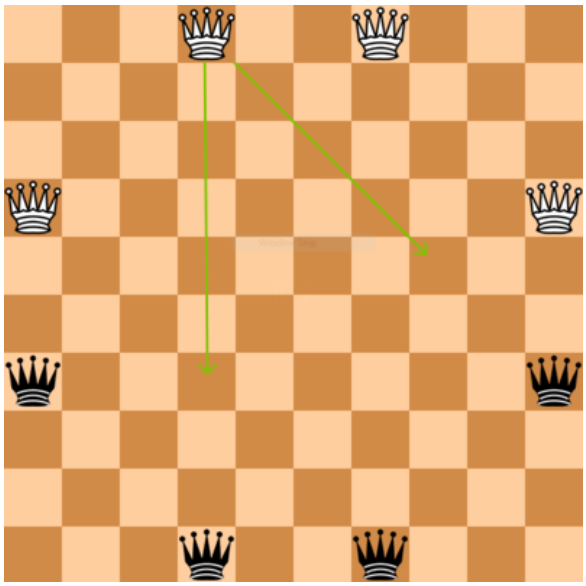

Abbildung 2.2: Zwei legale Züge die weiße Amazone zu bewegen ([12], eigene Bearbeitung)

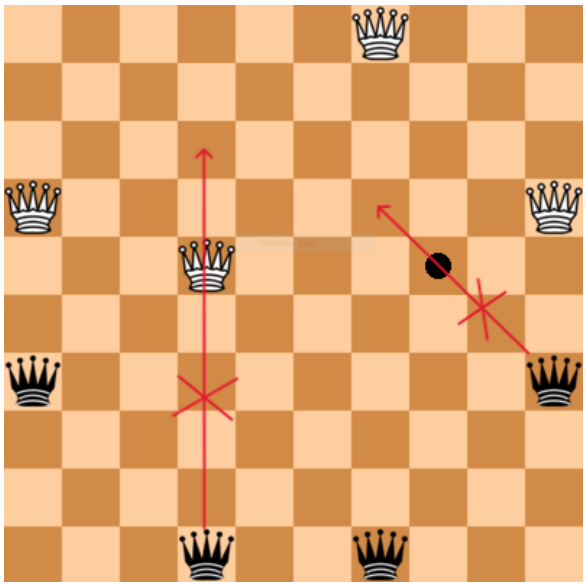

Abbildung 2.3: Zwei illegale Züge von Schwarz ([12], eigene Bearbeitung)

2. **Pfeil schießen**: Nachdem die Amazone auf ein leeres Feld gezogen wurde, muss sie in ebenfalls orthogonaler oder diagonaler Richtung einen Pfeil auf ein leeres Feld schießen. Dieser darf dabei auch keine andere Dame oder anderen Pfeil überqueren. Die Abbildungen 2.4 und 2.5 zeigen dabei wieder jeweils legale und illegale Züge für das Verschießen des Pfeils.

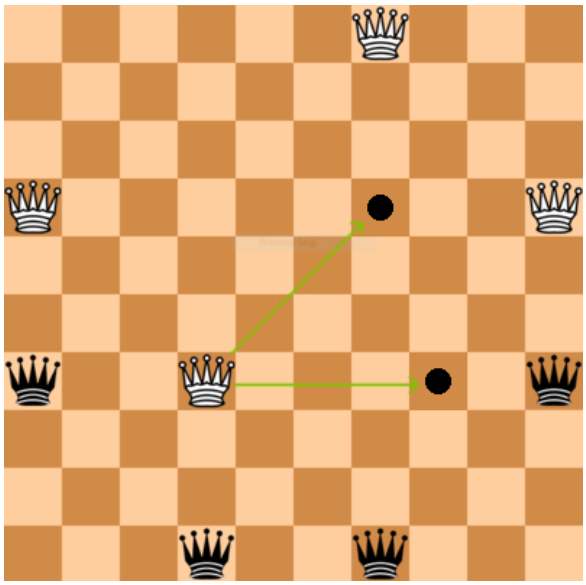

Abbildung 2.4: Zwei legale Möglichkeiten von der weißen Amazone den Pfeil zu schießen ([12], eigene Bearbeitung)

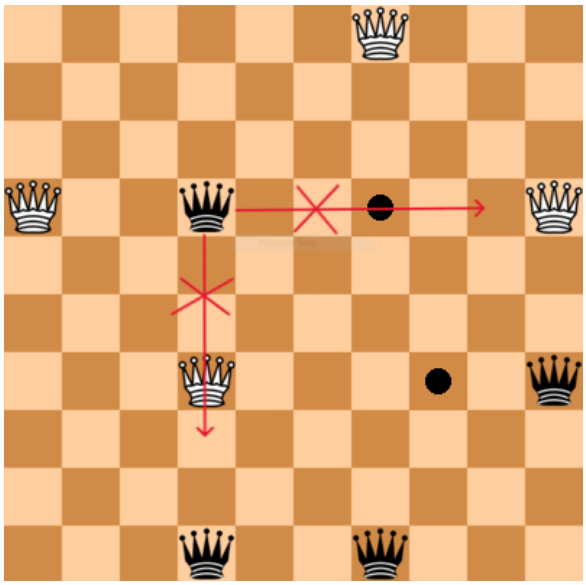

Abbildung 2.5: Zwei illegale Züge den Pfeil zu verschießen von Schwarz ([12], eigene Bearbeitung)

Da in Amazons eine **Zugpflicht** besteht, sprich der Spieler am Zug muss eine Amazone bewegen und einen Pfeil verschießen, verliert der Spieler, der als Erster einen Zug nicht mehr durchführen kann.

2.2 Strategie

Die Strategie des Spiels ist es, den gegnerischen Spieler so in seinen Zügen einzuschränken, dass dessen Mobilität begrenzt wird. Durch das Verschießen der Pfeile und das Bewegen der Amazonen enstehen so im späteren Spielverlauf abgegrenzte Gebiete. Welcher der beiden Spieler die größeren Gebiete im späteren Spielverlauf für sich beansprucht hat, ist im Vorteil. Denn jeder Zug verkleinert die verfügbare Spielfläche. Die Amazonen bewegen sich dann nur noch in den gebildeteten Gebieten, bis ein Spieler schließlich nicht mehr nachziehen kann.

Ein Spiel kann typischerweise in drei grobe Phasen unterteilt werden [16]:

1. **Frühe Spielphase**: *Mobilität* - Hierbei ist es wichtig, die Amazonen weg von dem Rand des Spielbrettes in das zentrale Spielfeld zu bewegen, um eine große Flexibilität zu gewährleisten. Dadurch wird die Gefahr ausgeschlossen, in die *Ecke* gedrängt zu werden.

2. **Mittlere Spielphase**: *Vorläufige Gebiete* - Gebiete werden langsam gebildet; die Mobilität sollte auch hier noch gewährleistet werden, jedoch mit zusätzlichem Fokus auf das Bilden von Gebieten.

3. **Endspiel**: *Gebiete Absichern* - Das Abschotten von Gebieten gegenüber dem Gegner ist hier der zentrale Punkt für diese Spielphase. Verhindern, dass der gegnerische Spieler Zugang zu den eigenen oder größeren Gebieten hat.

2.3 Eigenschaften

Das Spiel weist mehrere charakteristische Eigenschaften auf, die es zu einer einzigartigen Herausforderung im Bereich der Spieltheorie machen.

2.3.1 Spielende und Ergebnisbestimmung

Ein zentrales Merkmal des Spiels ist, dass es immer zu einem klaren Ergebnis kommt. Das Spiel endet, wenn ein Spieler keinen gültigen Zug mehr machen kann. Das bedeutet, es gibt keine Patt-Situationen oder Unentschieden, wie man sie in anderen Spielen wie Schach kennt. Ein Spieler gewinnt immer durch das Blockieren aller möglichen Züge des Gegners. Da mit jedem Zug ein Pfeil auf das Spielfeld gesetzt wird und dieser nicht entfernt werden kann, gibt es keine zyklischen Wiederholungen von Spielzuständen.

2.3.2 Vorteil durch den Startzug

Der Spieler mit den weißen Amazonen hat im Spiel einen leichten Vorteil, da er den ersten Zug ausführen darf. Dieser erste Zug gibt Weiß die Möglichkeit, frühzeitig die Kontrolle über zentrale Positionen auf dem Spielfeld zu übernehmen.

2.3.3 Verzweigungsfaktor und Komplexität

Amazons zeichnet sich durch einen extrem hohen Verzweigungsfaktor (von 500 [1]) aus. Zu Beginn des Spiels für Spieler 1 gibt es 2176 mögliche Züge, was das Spiel zu einer Herausforderung sowohl für menschliche Spieler als auch für Computerprogramme macht [1].

2.3.4 Keine zufälligen Elemente

Es ist ein deterministisches Spiel, bei dem alle Informationen öffentlich sind und keine zufälligen Elemente (wie Würfel oder verdeckte Karten) ins Spiel kommen. Jeder Zug ist komplett berechenbar und transparent.

2.3.5 Parallelen zu anderen Spielen

Das Spiel hat Parallelen zu Spielen wie Schach und Go. Vom Schach übernimmt es die Bewegungsregeln der Amazone. Vom Spiel Go übernimmt es das Konzept der Gebiets- oder Territoriumskontrolle.

3 Algorithmen zur Spielbaum-Suche

Suchalgorithmen spielen eine zentrale Rolle für das Durchsuchen von Spielbäumen. Das Ziel dieser ist es, aus einer Vielzahl von möglichen Zügen denjenigen auszuwählen, der mit einer hohen Wahrscheinlichkeit zum besten Ergebnis führt. Der Hauptaspekt ist unter anderem die effiziente Durchsuchung des Spielbaums, um innerhalb einer begrenzten, vorgegebenen Zeit möglichst viele Spielpositionen zu bewerten und so den optimalen Zug zu finden. Dies ist gerade bei Amazons wichtig, da der Spielbaum sehr komplex und groß ausfällt.

In dieser Arbeit werden vier bekannte und häufig verwendete Algorithmen, die zur Spielbaumsuche verwendet werden, vorgestellt: *Alpha-Beta*, *Principal Variation Search (PVS)*, *MTD(f)* und *Monte Carlo Tree Search (MCTS)*. Jeder dieser Algorithmen verfolgt einen unterschiedlichen Ansatz, um die Komplexität und Tiefe eines Spielbaumes zu bewältigen. Die jeweiligen Vor- und Nachteile sowie ihre Effizienz hängen stark von der Art des Spiels und der Struktur des Suchraums ab. Im Folgenden werden die Algorithmen detailliert beschrieben und ihre Funktionsweise im Kontext des Spiels *Amazons* erläutert.

3.1 Alpha-Beta

Der *Alpha-Beta Algorithmus*, auch bekannt als *Alpha-Beta-Pruning*, ist eine optimierte Variante des Minimax-Suchalgorithmus, der verwendet wird, um den bestmöglichen Zug in einem Spielbaum effizienter (als der Minimax-Suchalgorithmus) zu ermitteln. Während der Minimax-Algorithmus darauf abzielt, den bestmöglichen Zug für den aktiven Spieler (den Maximierer oder Minimierer - je nachdem welcher Spieler am Zug ist) zu finden, durchsucht er dabei den gesamten Spielbaum, was in vielen Spielen mit tiefen und breiten Bäumen zu einer enormen Rechenlast führt. Der Alpha-Beta Algorithmus adressiert dieses Problem, indem er Zweige des Spielbaums abschneidet (*Pruning* genannt), die sicher keine besseren Ergebnisse liefern können als bereits gefundene Werte. Dieses *Pruning* führt zu einer erheblichen Reduktion der Anzahl der zu untersuchenden Knoten und somit zu einer deutlichen Effizienzsteigerung, ohne das Endergebnis zu beeinflussen [14].

3.1.1 Funktionsweise

Der Alpha-Beta Algorithmus funktioniert ganz nach dem *Minimax Prinzip*, bei dem die Spieler abwechselnd versuchen, den für sie besten[1] relativen Wert im Spielbaum zu maximieren (für den Maximierer, dieser hat die weißen Figuren) oder zu minimieren (für den Minimierer - die schwarzen Figuren). In diesem Zusammenhang stehen die beiden Variablen α und β für die besten (zu einem bestimmten Zeitpunkt) berechneten Werte, die die Spieler erreichen können:

- α (Alpha): Repräsentiert den **höchsten Wert**, den der **Maximierer** bis zum aktuellen Punkt erreichen kann. Der **Maximierer** versucht, diesen Wert durch die Auswahl optimaler Züge so hoch wie möglich zu halten.
- β (Beta): Repräsentiert den **niedrigsten Wert**, den der **Minimierer** bis zum aktuellen Punkt erreichen kann. Der **Minimierer** versucht, diesen Wert durch seine Zugauswahl so niedrig wie möglich zu halten.

Während Alpha-Beta den Spielbaum durchläuft, werden jeweils die Alpha- und Beta-Werte entlang der jeweiligen Knoten aktualisiert. Wenn der aktuelle Spieler an einem Knoten feststellt, dass dieser keine besseren Ergebnisse erzielen kann, als die bereits bestehenden Alpha- oder Beta-Werte, wird der Zweig von diesem Knoten nicht weiter untersucht. Dieses *Pruning* sorgt also dafür, dass weitere unnötige Berechnungen durchgeführt werden müssten, da sicher ist, dass dieser Teilbaum für die Entscheidung des besten Zuges irrelevant ist.

Der Algorithmus durchläuft rekursiv die möglichen Spielzüge, bewertet diese und wählt den besten aus, um entweder das Spiel zu maximieren oder zu minimieren.

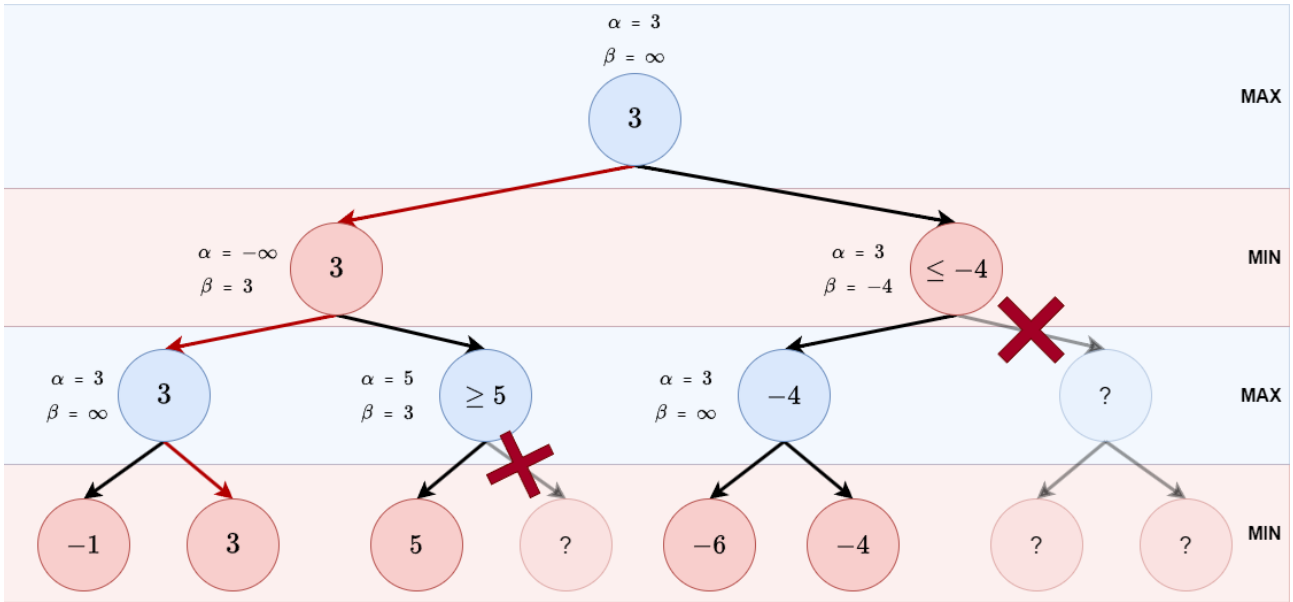

Abbildung 3.1: Alpha-Beta-Pruning

[1] Es wird vorausgesetzt, dass beide Spieler bei jedem Zug die für sie optimale Entscheidung treffen.

Erläuterung des Prunings anhand eines Beispiels

Das hier in der Abbildung 3.1 gezeigte Beispiel illustriert den Ablauf des Alpha-Beta-Prunings, der verwendet wird, um effizientere Entscheidungen in Minimax-Szenarien zu treffen, indem nicht notwendige Zweige des Spielbaums abgeschnitten werden. Dabei werden die Alpha- und Beta-Werte verwendet, um die Grenzen für MAX- und MIN-Knoten zu setzen, wodurch der Algorithmus frühzeitig erkennt, ob bestimmte Zweige keine besseren Ergebnisse liefern können und somit nicht weiter durchsucht werden müssen.

Der Algorithmus startet bei der Wurzel, die ein MAX-Knoten ist. Hier und bei den folgenden betrachteten Knoten werden die Alpha- und Beta-Werte zunächst auf $-\infty$ und $+\infty$ gesetzt. Im ersten Schritt wird der linke Kindknoten auf der MIN-Ebene untersucht. Der Algorithmus geht tiefer in den Baum und betrachtet die blauen MAX-Kindknoten dieses ersten roten MIN-Knotens. Der erste Kindknoten des blauen MAX-Knotens liefert den Wert -1, welcher in den darüberliegenden MAX-Knoten propagiert wird und $\alpha = -1$ gesetzt wird, da $-1 > \alpha$ und $\alpha = -\infty$ ist. Der zweite Knoten liefert den Wert 3 und da dieser größer als -1 ist, wird α auf 3 aktualisiert. Dieser Wert wird nun nach oben an den MIN-Knoten propagiert und $\beta = 3$ wird aktualisiert. Nun wird der andere MAX-Kindknoten des ersten MIN-Knoten betrachtet, dort wird der β Wert ebenfalls auf 3 aktualisiert. Dessen Kindknoten liefert den Wert 5. Die 5 wird wieder in α gespeichert, da $5 > -\infty$. Jetzt ist $\beta \leq \alpha$, wodurch der andere Kindknoten nicht weiter betrachtet werden muss. Der Minimierer hat hier bereits eine bessere Option, nämlich die 3. Der Minimierer würde bei dem ersten MIN-Knoten in jedem Fall die 3 wählen im Vergleich zu einem potentiellen Wert von ≥ 5, deshalb muss der Kindknoten nicht weiter betrachtet werden. Dieses Ergebnis wird wieder zurück an die Wurzel propagiert und α wird auf 3 gesetzt.

An diesem Punkt ist garantiert, dass der Maximierer eine Punktzahl von mindestens 3 erreichen kann. Jetzt wird jedoch der rechte Teil des Baumes untersucht, um zu gucken, ob ein besseres Ergebnis erzielt werden kann. Bei jedem folgenden Knoten wird $\alpha = 3$ und $\beta = +\infty$ als Information mit weitergegeben. Nun wird wieder der erste MIN-Knoten untersucht und weiter im Baum, der MAX-Kindknoten. Dessen Kindknoten liefern die Werte -6 und -4. Der α Wert ändert sich nicht, da die aktuelle Option 3 eine bessere für den Maximierer ist. Der Maximierer wählt den höheren Wert der Kindknoten aus und erhält so die -4, welche für β eingespeichert wird. Da nun $\beta \leq \alpha$ ist mit $\beta = -4$ und $\alpha = 3$ wird auch hier der Kindknoten nicht weiter untersucht, da auch hier in jedem Fall der Maximierer sich für die 3 entscheidet.

Das Beispiel zeigt, wie der Alpha-Beta-Algorithmus an zwei Stellen die Suche frühzeitig abbricht, da durch die Alpha- und Beta-Schranken erkannt wurde, dass bestimmte Zweige des Baumes nicht weiter untersucht werden müssen. Das Endergebnis bedeutet, dass der aktuelle Spieler einen Vorteil in der aktuellen Spielposition mit dem Wert 3 (gemäß der Bewertungsfunktion) erreichen kann.

Implementierung des Alpha-Beta-Algorithmus

Demnach lässt sich ein typischer Ablauf des Alpha-Beta Algorithmus (bereits mit einer Optimierung durch eine Transpositionstabelle) folgendermaßen in Pseudo-Code darstellen:

Algorithm 1 Alpha-Beta-Pruning mit Transposition Tables

```
1: Function AlphaBeta(board, depth, alpha, beta, maximizingPlayer)
2: if depth == 0 or timeout or GameIsOver() then
3:     return EvaluationFunction(CurrentState)
4: end if
5: board_hash = hash_board(board)
6: if board_hash is found in transposition_table then
7:     stored_depth, stored_eval, stored_move = transposition_table[board_hash]
8:     if stored_depth ≥ depth then
9:         return stored_eval, stored_move
10:    end if
11: end if
12: best_move = None
13: if maximizingPlayer then
14:    maxEval = −∞
15:    for each PossibleMoveFromCurrentPlayer() do
16:        board.MakeMove(move)
17:        eval = AlphaBeta(board, depth - 1, alpha, beta, false)
18:        board.UndoMove(move)
19:        if eval > maxEval then                ▷ Aktueller Zug hat neue höchste Bewertung
20:            maxEval = eval
21:            best_move = move
22:        end if
23:        alpha = max(alpha, eval)
24:        if beta ≤ alpha then
25:            break                                                    ▷ Beta-Cutoff
26:        end if
27:    end for
28:    transposition_table[board_hash] = (depth, maxEval, best_move)
29:    return maxEval, best_move
30: else
31:    minEval = +∞
32:    for each PossibleMoveFromCurrentPlayer() do
33:        board.MakeMove(move)
34:        eval = AlphaBeta(board, depth - 1, alpha, beta, true)
35:        board.UndoMove(move)
36:        if eval < minEval then                ▷ Aktueller Zug hat neue niedrigste Bewertung
37:            minEval = eval
38:            best_move = move
39:        end if
40:        beta = min(beta, eval)
41:        if beta ≤ alpha then
42:            break                                                    ▷ Alpha-Cutoff
43:        end if
44:    end for
45:    transposition_table[board_hash] = (depth, minEval, best_move)
46:    return minEval, best_move
47: end if
```

3.1.2 Effizienzsteigerung durch zusätzliche Techniken

Um den Algorithmus noch effizienter und spielstärker zu gestalten wurden fortgeschrittene Techniken wie **Transposition Tables** und **Iterative Deepening** verwendet.

Transpositionstabelle

Eine **Transpositionstabelle** (englisch: Transposition Tables) ist eine wichtige Optimierungstechnik in Spielbaumsuchalgorithmen wie Minimax oder Alpha-Beta-Pruning, die bei Spielen wie Schach oder Dame verwendet werden. Komplexe Spiele wie Amazons haben eine riesige Anzahl von möglichen Zuständen (Züge und Brettkonfigurationen), die untersucht werden müssen, um die beste Entscheidung zu treffen. Dadurch können viele dieser Zustände während der Berechnungen mehrfach auftreten und Transpositionstabellen können den Algorithmus erheblich beschleunigen, indem sie solche wiederholte Berechnungen verhindern.

Die Transpositionstabelle ist eine Hash-basierte Datenstruktur (meist eine Hash-Map oder ein ähnliches Konstrukt), die die Bewertung von bereits untersuchten Spielzuständen zwischenspeichert. Der Algorithmus muss also denselben Zustand im Suchbaum nicht mehrmals analysieren und berechnen, sondern kann die bereits berechnete und abgespeicherte Stellung in der Transpositionstabelle abgreifen. Dadurch wird sowohl Zeit als auch Rechenleistung gespart.

Die Transpositionstabelle speichert:

- **Den Hash des Brettzustands:** Dieser Hash dient als eindeutiger Schlüssel, der den aktuellen Spielzustand repräsentiert.
- **Bewertung des Zustands:** Dies ist der numerische Wert, der die Bewertung dieses Zustands darstellt, basierend auf der Bewertungsfunktion.
- **Tiefe der Suche:** Die Tiefe, auf der der Zustand berechnet wurde, ist ebenfalls wichtig, da eine höhere Tiefe eine präzisere Bewertung liefert. Wenn der Zustand bereits auf einer größeren Tiefe berechnet wurde, ist es besser, diese Bewertung zu verwenden.

Beim Durchlaufen des Spielbaums sucht der Algorithmus in jeder Knotenposition in der Transpositionstabelle nach einem bereits gespeicherten Ergebnis für diesen Zustand. Der Algorithmus funktioniert folgendermaßen:

- **Suchen:** Zu Beginn der Berechnung eines neuen Knotens überprüft der Algorithmus, ob der Zustand des Spielbretts bereits in der Transpositionstabelle gespeichert ist. Sprich ob der derzeitige Hash-Wert der aktuellen Spielposition in der Transpositionstabelle vorhanden ist.
- **Vergleichen der Tiefe:** Falls der Zustand in der Tabelle ist, wird überprüft, ob der gespeicherte Wert auf einer größeren oder gleichen Tiefe berechnet wurde als der aktuelle Knoten. Wenn dies der Fall ist, wird dieser bereits gespeicherte Wert verwendet.
- **Speichern:** Wenn ein Zustand noch nicht in der Transpositionstabelle ist oder auf einer höheren Tiefe berechnet wurde, wird er nach der Berechnung in die Tabelle eingefügt oder der gespeicherte Wert aktualisiert.

Dies spart nicht nur Zeit, sondern ermöglicht es auch, Knoten auf höheren Tiefen genau zu bewerten, wodurch Alpha-Beta effizienter wird.

Bei der Implementierung der Transpositionstabelle wird typischerweise das klassische **Zobrist Hashing** verwendet, aufgrund seiner Schnelligkeit und einfachen Implementierung. Nach einigen Tests stellte sich jedoch heraus, dass der Einsatz des **SHA-256** Algorithmus als Hash-Funktion keinerlei spürbaren Unterschied in der Berechnungszeit mit sich brachte und ebenfalls einfach zu implementieren ist.

SHA-256 hat den Vorteil einer starken "Kollisionseigenschaft", was das Risiko minimiert, dass zwei verschiedene Brettzustände denselben Hash-Wert erhalten und somit möglicherweise falsch als identisch interpretiert werden. In der Praxis ist Zobrist Hashing zwar oft die präferierte Methode, da es speziell für Spiele wie Schach entwickelt wurde, jedoch hat SHA-256 in dieser Arbeit keine negativen Auswirkungen auf die Performance gezeigt.

Iterative Deepening

Iterative Deepening ist eine Suchstrategie, die häufig in Spielen wie Schach oder andere komplexen Spiele eingesetzt wird. Dabei werden Elemente der Tiefensuche (Deph-First Search) und Breitensuche (Breadth-First Search) kombiniert, indem die Suche schrittweise mit einer zunehmenden Tiefe wiederholt wird. Diese Suche wird solange durchgeführt bis entweder eine gewünschte maximale Suchtiefe erreicht wurde oder eine festgelegte Zeitbegrenzung überschritten wurde. Korf zeigte, dass Iterative Deepening durch die schrittweise Vertiefung der Suche in Spielen mit großem Suchbaum eine effiziente und optimale Lösung für das Finden des besten Zuges bietet [4].

Der zentrale Vorteil von Iterative Deepening für die Algorithmen liegt darin, dass nach jeder durchlaufenen Suchtiefe ein vollständiges Ergebnis für die jeweilige Suchtiefe vorliegt. Dieses Ergebnis kann als eine Art Zwischenlösung betrachtet werden. Das ist besonders wichtig bei Spielen in denen es eine Zeitbegrenzung gibt, da so bei Überschreitung der Zeit, dennoch ein Zwischenergebnis (in Form des besten bislang gefundenen Zuges) vorliegt [4]. Die Nutzung der Transpositionstabelle ermöglicht es zudem, bereits berechnete Spielpositionen zu speichern und so in späteren Iterationen des Iterative Deepenings wiederzuverwenden. Diese Methode minimiert den Speicherbedarf und erlaubt durch die Wiederverwendung von berechneten Spielpositionen aus Transpositionstabellen eine hohe Effizienz [8].

Der Algorithmus startet typischerweise bei einer Suchtiefe von eins und erhöht diese dann um eins, nachdem die jeweilige Suchtiefe abgeschlossen ist. Der beste Zug wird nach jeder vollständig beendeten Iteration aktualisiert und abgespeichert.

3.2 Principal Variation Search (PVS)

Principal Variation Search (PVS) ist eine Verfeinerung des *Alpha-Beta Suchalgorithmus*, die speziell darauf abzielt, die Suche effizienter zu gestalten, indem sie eine tiefere Analyse der sogenannten *Principal Variation* (der bestmöglichen Zugsequenz) ermöglicht. PVS basiert auf der Annahme, dass der erste Zug in einer bestimmten Position in der Regel der beste ist, sodass der Algorithmus versucht,

diesen zuerst vollständig zu durchsuchen, bevor weitere Züge in Betracht gezogen werden [11][10]. In Fällen, in denen der erste Zug tatsächlich der beste ist, kann der Algorithmus effizienter agieren, da weniger Berechnungen notwendig sind, um alternative Züge zu evaluieren. Dies führt zu einer signifikanten Verringerung der Rechenzeit im Vergleich zu einer vollständigen *Alpha-Beta* Suche.
Der Hauptunterschied zur herkömmlichen *Alpha-Beta* Suche besteht darin, dass PVS für alle Züge außer dem ersten eine reduzierte Suche (*Null Window Search*) verwendet, um zu prüfen, ob der Zug den bereits gefundenen besten Wert (*Principal Variation*) übertrifft. Nur wenn dies der Fall ist, wird der Zug vollständig durchsucht, um zu überprüfen, ob er tatsächlich besser ist.

PVS kann ebenfalls, durch die Kombination mit Techniken wie **Transposition Tables** und **Iterative Deepening** noch effizienter gestaltet werden.

Implementierung des PVS-Algorithmus

Der PVS Algorithmus kann als folgender Pseudo-Code dargestellt werden:

Algorithm 2 Principal Variation Search (PVS) - Teil 1 (Maximizing Player)

```
1:  Function PVS(board, depth, alpha, beta, maximizing_player, best_move_prev=None)
2:  if depth = 0 or timeout or GameIsOver() then
3:      return EvaluationFunction(CurrentState)
4:  end if
5:  /* Board hash logic here */
6:  best_move ← None
7:  first_move ← True
8:  if maximizing_player = True then
9:      max_eval ← −∞
10:     moves ← [best_move_prev] + PossibleMovesFromCurrentPlayer()
11:     for move in moves do
12:         board.MakeMove(move)
13:         if first_move = True then
14:             eval ← PVS(board, depth - 1, alpha, beta, False)          ▷ Volle Fenster-Suche
15:             first_move ← False
16:         else
17:             eval ← PVS(board, depth - 1, alpha, alpha + 1, False)     ▷ Null Fenster-Suche
18:             if eval > alpha then
19:                 eval ← PVS(board, depth - 1, alpha, beta, False)      ▷ Volle Fenster-Suche
20:             end if
21:         end if
22:         board.UndoMove(move)
23:         if eval > max_eval then
24:             max_eval ← eval
25:             best_move ← move
26:             if timeout then
27:                 transposition_table[board_hash] ← (depth, max_eval, best_move)
28:                 return (max_eval, best_move)
29:             end if
30:         end if
31:         alpha ← max(alpha, eval)
32:         if beta ≤ alpha then
33:             Break the loop
34:         end if
35:     end for
36:     transposition_table[board_hash] ← (depth, max_eval, best_move)
37:     return (max_eval, best_move)
38: end if
```

Algorithm 3 Principal Variation Search (PVS) - Teil 2 (Minimizing Player)

```
1:
2:  if maximizing_player = False then
3:      min_eval ← ∞
4:      moves ← [best_move_prev] + PossibleMovesFromCurrentPlayer()
5:      for move in moves do
6:          board.MakeMove(move)
7:          if first_move = True then
8:              eval ← PVS(board, depth - 1, alpha, beta, True)          ▷ Volle Fenster-Suche
9:              first_move ← False
10:         else
11:             eval ← PVS(board, depth - 1, beta - 1, beta, True)       ▷ Null Fenster-Suche
12:             if eval < beta then
13:                 eval ← PVS(board, depth - 1, alpha, beta, True)      ▷ Volle Fenster-Suche
14:             end if
15:         end if
16:         board.UndoMove(move)
17:         if eval < min_eval then
18:             min_eval ← eval
19:             best_move ← move
20:             if timeout then
21:                 transposition_table[board_hash] ← (depth, min_eval, best_move)
22:                 return (min_eval, best_move)
23:             end if
24:         end if
25:         beta ← min(beta, eval)
26:         if beta ≤ alpha then
27:             Break the loop
28:         end if
29:     end for
30:     transposition_table[board_hash] ← (depth, min_eval, best_move)
31:     return (min_eval, best_move)
32: end if
```

Null-Fenster-Suche

Die Null-Fenster-Suche ist das zentrale Element des PVS-Algorithmus, bei dem das Suchfenster auf eine *Null-Breite* reduziert wird. Dabei wird zunächst nur geprüft, ob ein Zug besser als der aktuelle beste Zug ist. Wenn das der Fall ist, wird für diesen Zug eine vollständige Alpha-Beta-Suche durchgeführt. Im anderen Fall wird mit dem nächsten Zug fortgefahren und der aktuell betrachtete Zug wird übersprungen. Dadurch werden viele unnötige Suchen vermieden, da oft der erste untersuchte Zug in der Liste der möglichen Züge der beste ist.

Die Implementierung der Null-Fenster-Suche (für den Maximierer) sieht wie folgt aus:

Listing 3.1: Null-Fenster-Suche in PVS (für den Maximierer)

```
if first_move:
    # Vollständige Suche
    eval, _ = pvs(game, depth-1, alpha, beta, False)
    first_move = False
else:
    # Null-Fenster-Suche
    eval, _ = pvs(game, depth-1, alpha, alpha+1, False)
    if eval > alpha:
        # Vollständige Suche falls nötig
        eval, _ = pvs(game, depth-1, alpha, beta, False)
```

Im oben gezeigten Code wird der erste Zug (if first_move) immer mit einem vollständigen Alpha-Beta-Suchfenster (alpha, beta) überprüft. Für alle weiteren Züge (else-Klausel) wird eine Null-Fenster-Suche mit:

- (alpha, alpha+1) für den Maximierer
- (beta-1, beta) für den Minimierer

durchgeführt und falls diese Suche ergibt, dass der Zug besser als die aktuelle Alpha-Beta-Bewertung ist (für den Minimierer if eval < beta), wird eine erneute vollständige Suche gestartet, um die genaue Bewertung des Zuges zu ermitteln.

Zug-Sortierung mit dem besten Zug der vorherigen Iteration

Da der PVS-Algorithmus davon ausgeht, dass der erste Zug in der Liste der möglichen Züge der beste ist, sollte jeweils der beste Zug aus der vorherigen **Iterative Deepening**-Iteration an den Anfang der Liste gesetzt werden.

Listing 3.2: Zug-Sortierung im PVS

```
moves = []
if best_move_prev:
    moves.append(best_move_prev)
for move in game.calculate_moves_for_current_player():
    if move != best_move_prev:
        moves.append(move)
```

Durch diesen Ansatz wird gewährleistet, dass der in der vorherigen Iteration als optimal bewertete Zug zuerst überprüft wird und dadurch die Wahrscheinlichkeit erhöht, dass der erste Zug tatsächlich der beste ist.

3.3 MTD(f)

MTD(f), was für *Memory-enhanced Test Driver with a guess function* steht, ist ein hochoptimierter Minimax-Suchalgorithmus, der von Aske Plaat et al. entwickelt wurde [3]. MTD(f) nutzt Zero-

Window-Suchen (Nullfenster), um eine schnelle Konvergenz zum optimalen Zug zu erreichen. Anstelle eines breiten Suchfensters, wie es bei der klassischen Alpha-Beta-Suche eingesetzt wird, verwendet MTD(f) enge Nullfenster, die lediglich obere oder untere Schranken des Suchbereichs zurückgeben. Durch diese Technik kann der Algorithmus durch häufigeres Abschneiden die Effizienz deutlich steigern, auch wenn dadurch mehr Wiederholungen erforderlich sind [7].

MTD(f) wurde ursprünglich für Spielprogramme wie Schach, Dame oder Othello entwickelt und hat sich in verschiedenen Tests als leistungsfähiger erwiesen als herkömmliche Ansätze [3]. Dies liegt daran, dass MTD(f) auf der wiederholten Anwendung der Alpha-Beta-Suche aufbaut und sich schrittweise an den exakten Minimax-Wert annähert, indem in jeder Iteration der Suchbereich angepasst wird. In Amazons zeigt MTD(f) durch die Nutzung minimaler Fenster, dass es im Vergleich zur Alpha-Beta- und Negamax-Suche eine höhere Effizienz und schnellere Berechnungen erreicht [11].

Implementierung und Pseudo-Code

Die Implementierung von MTD(f) erfolgt in Kombination mit der Alpha-Beta-Suche, wobei Nullfenster als Suchfenster genutzt werden. Der folgende Pseudo-Code veranschaulicht die Implementierung der *MTD(f)*-Funktion:

Algorithm 4 MTD(f)-Algorithmus

```
1:  function MTD_F(board, f, d, maximizing_player)
2:      g ← f
3:      upper_bound ← ∞
4:      lower_bound ← −∞
5:      while lower_bound < upper_bound do
6:          if g = lower_bound then β ← g + 1 else β ← g
7:          g, best_move ← AlphaBeta_With_Null_Window(board, d, β − 1, β, maximizing_player)
8:          if g < β then
9:              upper_bound ← g
10:         else
11:             lower_bound ← g
12:         end if
13:     end while
14:     return g, best_move
15: end function
```

Implementierung der Transpositionstabelle

Wie andere Algorithmen der Spielbaum-Suche profitiert auch MTD(f) von Techniken wie **Transposition Tables** und **Iterative Deepening**. Das Speichern der Transpositionstabelle für den MTD(f)-Algorithmus unterscheidet sich von den anderen Algorithmen. Hier werden zusätzlich sogenannte *Flags* verwendet, um die Schranken des gespeicherten Werts zu kennzeichnen. Für jeden gespeicherten Eintrag wird ein Flag gesetzt, das angibt, ob der gespeicherte Wert genau (*EXACT*), eine Untergrenze (*LOWERBOUND*) oder eine Obergrenze (*UPPERBOUND*) ist. Bei der Wiederverwendung eines Eintrags prüft der Algorithmus anhand dieser Flags, ob der gespeicherte Wert für die aktuellen

Alpha- und Beta-Schranken direkt verwendet werden kann. Im folgenden wird nur die Implementierung der Transpositionstabelle als Pseudo-Code dargelegt:

Algorithm 5 Alpha-Beta-Suche mit Nullfenster und Transpositionstabelle

```
1:  function AlphaBeta_With_Null_Window(board, d, α, β, maximizing_player)
2:      board_hash ← hash_board(board)
3:      entry ← transposition_table.get(board_hash)
4:      if entry ≠ None and entry.depth ≥ d then
5:          if entry.flag = EXACT then
6:              return entry.value
7:          else if entry.flag = LOWERBOUND and entry.value ≥ β then
8:              return entry.value                                  ▷ Beta-Cutoff
9:          else if entry.flag = UPPERBOUND and entry.value ≤ α then
10:             return entry.value                                  ▷ Alpha-Cutoff
11:         end if
12:     end if
13:     Führe Alpha-Beta-Suche durch und speichere Ergebnisse in der Transpositionstabelle
14: end function
```

Die Flags werden dabei am Ende der Alpha-Beta Suche (vor Ende der jeweiligen Tiefensuche) wie folgt gespeichert:

Listing 3.3: Speichern der Flag für Hash-Eintrag

```
flag = LOWERBOUND if max_eval >= beta
else (UPPERBOUND if max_eval <= alpha else EXACT)
```

3.4 Monte Carlo Tree Search

Der Monte Carlo Tree Search (MCTS) Algorithmus ist eine stochastische Suchmethode, die in den letzten Jahren durch ihre Erfolge in Spielen wie Go und Amazons an Bedeutung gewonnen hat [1]. MCTS kombiniert eine Baumsuchstrategie mit Monte Carlo-Simulationen (auch als *Playouts* bezeichnet), wobei die Ergebnisse dieser Simulationen verwendet werden, um Zustände in einem Suchbaum zu bewerten. Diese Methode ermöglicht es, eine Strategie ohne vorher festgelegte Bewertungsfunktion zu entwickeln, was besonders in Spielen mit einem hohen Zustandsraum, wie Go und Amazons, von Vorteil ist [9].

Der MCTS-Algorithmus hat dabei folgende grundlegende Komponenten:

- **Knoten:** Ein Knoten repräsentiert eine Spielposition innerhalb des Baumes. Jeder Knoten enthält Informationen darüber, wie oft er besucht wurde und wie viele Siege die nachfolgenden Knoten erzielen konnten.
- **Kanten:** Die Kanten zwischen den Knoten repräsentieren mögliche Züge der Spieler. Sie verbinden Eltern- und Kindknoten, wodurch die möglichen Zugfolgen im Baum dargestellt werden.
- **Besuche (Visits):** Jeder Knoten speichert die Anzahl der Male, die er im Verlauf der Suche besucht wurde.
- **Gewinne (Wins):** Die Anzahl der Gewinne wird für jeden Knoten festgehalten, basierend auf den Ergebnissen der Simulationen, die durch diesen Knoten geführt haben.

Die vier Phasen des MCTS-Algorithmus (siehe Abbildung 3.2) [13, 14]:

- **Selection**: Ausgehend von der Wurzel wird entlang des Baumes ein Knoten basierend auf einem Auswahlkriterium (beispielsweise Upper Confidence Bound applied to Trees (UCT)) selektiert, bis ein Knoten ohne Kindknoten (auch *Blattknoten*) erreicht ist.
- **Expansion**: Sobald solch ein Blattknoten erreicht ist, wird dieser um einen zufällig gewählten Kindknoten erweitert.
- **Simulation**: Für diesen neuen Kindknoten wird eine zufällige Simulation des Spiels (*Playouts*) bis zu einer selbst festgelegten Anzahl der Simulationen oder Spielende durchgeführt. Diese Simulationen liefern eine Schätzung des Wertes der Spielsituation.
- **Backpropagation**: Die Ergebnisse der Simulation werden durch den Baum zurückgetragen, um die Werte der Knoten zu aktualisieren. Dabei wird der Wert des aktuellen Knotens auf Basis der Simulationsergebnisse angepasst.

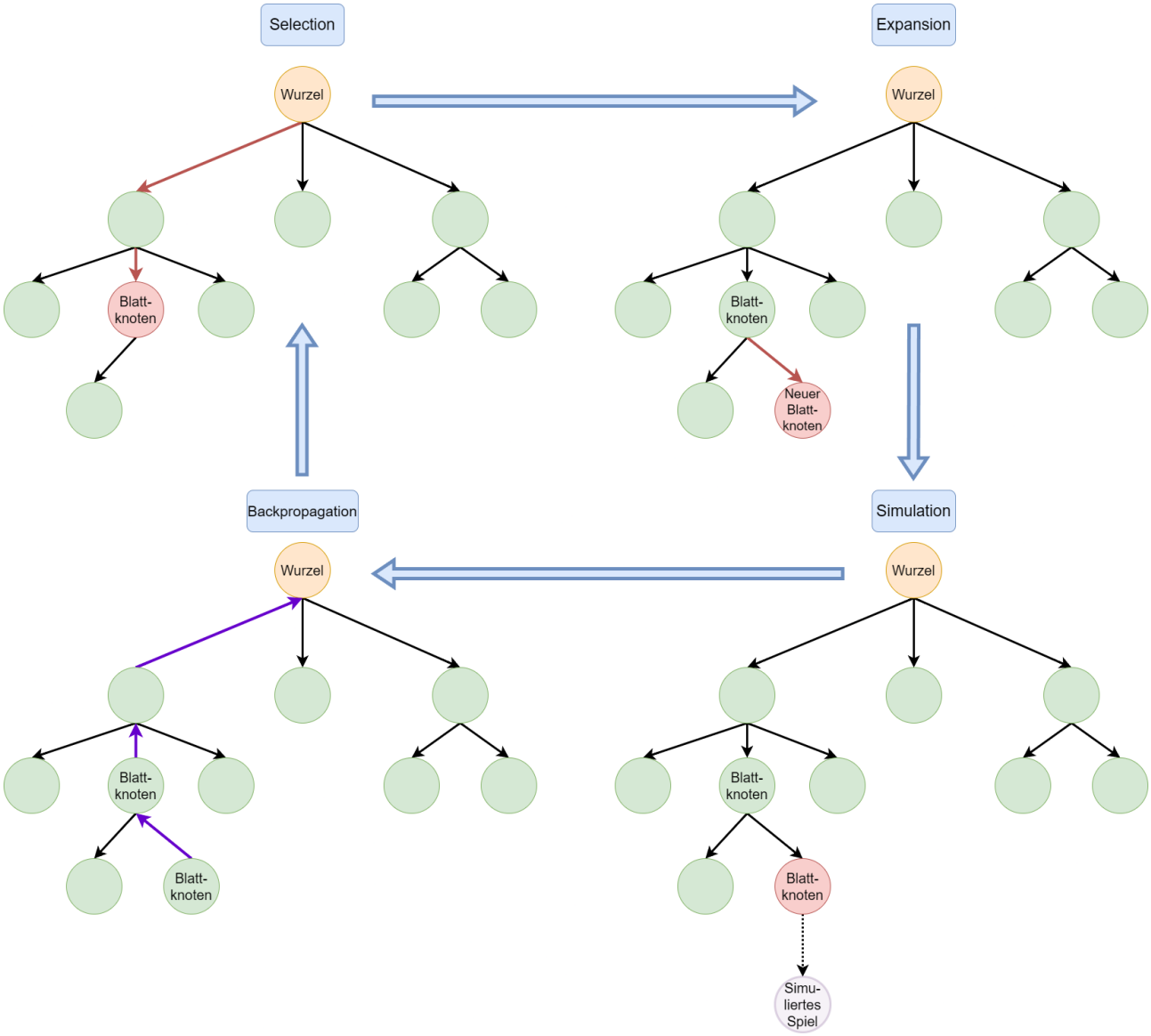

Abbildung 3.2: Phasen des Monte Carlo Tree Search Algorithmus

Jede Phase trägt dazu bei, den Suchbaum effizient aufzubauen und die beste nächste Aktion zu finden [9]. Der Algorithmus wählt in der Selektionsphase wiederholt Knoten aus, um tiefer in den Suchbaum zu gehen. Sobald ein Blattknoten erreicht wird, wird dieser expandiert, und es wird ein neuer Knoten hinzugefügt. Im Simulationsschritt werden zufällige Playouts durchgeführt, bis eine gewünschte Anzahl von Simulationen erreicht oder das Spiel vorbei ist. Schließlich wird das Ergebnis der Simulation in der Rückpropagationsphase auf die besuchten Knoten zurückgeführt (dabei erhöht sich auch die Besuch-Anzahl um eins), um die Bewertung der vorherigen Zustände zu aktualisieren [1].

3.4.1 Upper Confidence Bound applied to Trees (UCT)

Die Auswahl der Kinderknoten während der Baumsuche ist besonders entscheidend für den Verlauf des Algorithmus. Der Upper Confidence Bound applied to Trees (UCT)-Algorithmus dient zu einer

geeigneten und sinnvollen Auswahl solcher [14]. Dieser Ansatz balanciert die Exploration und die Exploitation optimal aus. Exploration und Exploitation lassen sich wie folgt definieren:

- **Exploration** steht für den Teil des Algorithmus, bei dem bislang wenig oder kaum untersuchte Züge dennoch untersucht werden. Bedeutet, das bewusst Züge gewälht werden, die in früheren Iterationen seltener untersucht wurden, sodass mehr Informationen über deren Potenzial gesammelt wird. Dadurch wird sichergestellt, dass potenziell gute Züge nicht übersehen werden, die möglicherweise erst nach tieferer Betrachtung einen guten Zug darstellen.

- **Exploitation** dagegen ist der Teil des Algorithmus, der gute und bereits bekannte Züge bevorzugt, die ohnehin schon als vielversprechend erscheinen. Um dies zu beeurteilen verwendet der Algorithmus die Ergebnisse aus früheren Simulationen oder Berechnungen und wählt dabei den Zug aus, der in der Vergangenheit die besten Ergebnisse erzielt hat. Dadurch wird gewährleistet, dass die Zugauswahl auf dem bereits Gelernten basiert.

Die Formel zur Berechnung des UCT-Wertes eines Knoten i lautet:

$$UCT(i) = \bar{X}_i + C\sqrt{\frac{2 * \ln(n_p)}{n_i}}$$

wobei:

- $\bar{X}_i$ die **Gewinnrate** des Knotens i ist und durch das Verhältnis der Siege eines Kindknotens zur Anzahl der Besuche dieses Knotens definiert ist:

  $$\bar{X}_i = \frac{\text{Siege\_des\_Kindknotens}_i}{\text{Besuche\_des\_Kindknotens}_i}$$

  Dieser Term steht für **Exploitation**, da er den aktuellen „Wert“ des Knotens basierend auf den bisherigen Simulationen angibt. Je höher dieser Wert ist, desto mehr wird dieser Knoten in Zukunft bevorzugt, da er in der Vergangenheit erfolgreich war.

- C ein **Explorationsparameter** ist, der die Gewichtung der Exploration bestimmt. Je höher C ist, desto stärker wird die Exploration gefördert. Ein kleiner Wert für C würde die Ausnutzung der bereits gesammelten Informationen bevorzugen.

- n_p die Gesamtanzahl der Besuche des **Elternknotens** p repräsentiert. Der Term $\ln(n_p)$ wächst, je öfter der Elternknoten besucht wird, und sorgt dafür, dass die Exploration mit zunehmenden Simulationen abnimmt.

- n_i die Anzahl der Besuche des Knotens i ist. Wenn ein Knoten seltener besucht wurde, sorgt der **Explorationsterm**

  $$\sqrt{\frac{2 * \ln(n_p)}{n_i}}$$

  dafür, dass dieser Knoten in zukünftigen Simulationen häufiger ausgewählt wird. Der Explorationsterm wird umso größer, je kleiner n_i ist, was den Algorithmus dazu anregt, weniger erforschte Knoten zu besuchen.

Zusätzlich zur Standardberechnung des UCT-Wertes wird in dieser Arbeit die **move_priority** eingeführt, die die Bewertung des resultierenden Spielzustands eines Zuges berücksichtigt. Diese Priorisierung der Züge basiert auf der Bewertung der Spielsituation, die durch die Bewertungsfunktion des Spiels berechnet wird. Hierbei handelt es sich um eine Anpassung der Standard-UCT-Berechnung, um den algorithmischen Fortschritt noch genauer zu steuern.

Die Gewichtung wird durch die Berechnung eines **adjusted_value** modifiziert, welcher sowohl den UCT-Wert als auch die *move_priority* berücksichtigt. Dies geschieht durch folgende Modifikation:

$$adjusted\_value = UCT(i) + \frac{move\_priority}{1 + n_i}$$

für den ersten Spieler und

$$adjusted\_value = -UCT(i) - \frac{move\_priority}{1 + n_i}$$

für den zweiten Spieler.

Diese Anpassung beeinflusst die Auswahl des besten Kindes insofern, dasss nicht nur die Balance zwischen Exploration und Exploitation berücksichtigt wird, sondern auch grundlegend starke Züge, die die relative Position im Spiel stärken, indem die Bewertung der resultierenden Spielposition einfließt. Das ist besonders hilfreich für den Anfang der Suche, denn die Gewichtung der **move_priority** nimmt mit der Anzahl der Besuche für den Knoten $(1 + n_i)$ ab. Züge, die also nach Bewertung der Spielposition vorteilhafter sind, werden so häufiger zu Beginn jeden Durchlaufs ausgewählt und im späteren Verlauf verliert die Anpassung immer mehr an Einfluss auf die Auswahl der Knoten. Für den zweiten Spieler wird der Wert der Bewertung subtrahiert, da dieser ein gegensätzliches Spielziel verfolgt (negative Werte der Bewertungsfunktion bedeuten einen Spielvorteil für Spieler 2). Dadurch wird der MCTS vorallem in der Anfangsspielphase für das Auswählen starker Knoten unterstützt.

Listing 3.4: Auswahl des Knoten mit *adjusted_value*

```
# adjusted_values werden gespeichert
choices_weights.append(adjusted_value)
# Als Spieler 1
return self.children[choices_weights.index(max(choices_weights))]
# Als Spieler 2
return self.children[choices_weights.index(min(choices_weights))]
```

Dementsprechend wählt der **Spieler 1** am Ende den Knoten mit dem **größten** berechneten Gewicht und **Spieler 2** den Knoten mit dem **kleinsten** berechneten Gewicht. Zu Beginn des UCT werden alle Knotengewichte des Spieler 1 auf $+\infty$ initialisiert und die des Spieler 2 auf $-\infty$ und dann jeweils im weiteren Verlauf durch den MCTS aktualisiert.

3.4.2 MCTS im Spiel Amazons

Da das Spiel eine hohe Anzahl an möglichen Zügen pro Zug besitzt (2176 mögliche Züge für Spieler 1, einem Branching-Faktor von circa 500) stellt das eine besondere Herausforderung für den MCTS dar, was die Suche nach optimalen Zügen erschwert [1]. Durch die Kombination von MCTS mit einer

geeigneten Bewertungsfunktion und anderen Modifikationen, kann die Spielstärke deutlich gesteigert werden. Die Implementierung von MCTS bei Amazons nutzt das UCT-Verfahren, um den Suchbaum zu erweitern und durch Playouts eine gute Abschätzung des nächsten besten Zuges zu erreichen [14].

Ein interessantes Experiment zeigt, dass die Erhöhung der Anzahl von Simulationen in einem MCTS-Programm nicht zwangsläufig zu einer Verbesserung der Spielstärke führt, insbesondere im Vergleich zu klassischen Alpha-Beta-Pruning-Verfahren [14]. Die Ergebnisse deuten darauf hin, dass eine Kombination von MCTS mit einer Bewertungsfunktion effektiver sein kann als das bloße Erhöhen der Simulationen. Dies zeigt, dass die Bewertungsfunktion eine zentrale Rolle im MCTS-basierten Spiel für Amazons spielt.

Parameter für den MCTS

Die Wahl der richtigen Parameter hängt von der jeweiligen Implementation und dem Spiel ab. Im späteren Kapitel 5.1 wird sich damit näher beschäftigt und versucht herauszufinden, welche Parameter die besten Ergebnisse erzielen. Als Ausgangspunkt wurde zunächst der Explorationswert C auf 1.4 gesetzt, um mehr potenziell starke Züge zu untersuchen, die zuerst nicht allzu vielversprechend erscheinen.

Normalerweise wird für den MCTS die Simulation bis zum Spielende durchgeführt. Aufgrund der Zeitbegrenzung wird hier die Simulation jedoch nach einer bestimmten Anzahl von Zügen gestoppt. Hierfür wurde zu Beginn mit einer Anzahl von 6 Zügen pro Simulation getestet (wie in der Arbeit von R. J. Lorentz [6]). Im späteren Verlauf wurden auch hierfür andere Werte für die Anzahl der Züge pro Simulation getestet (siehe Kapitel 5.1).

3.4.3 Verbesserungen des MCTS

Zusätzliche Optimierungen wie das Einfügen von Wissen in die zufälligen Spiele (*Playouts*) haben ebenfalls eine deutliche Verbesserung der Spielstärke zur Folge [2]. Beispielsweise kann die sogenannte *Liberty Rule* verwendet werden, bei der die Mobilität der Figuren berücksichtigt wird, um strategische Vorteile zu maximieren [1]. Darüber hinaus kann durch das Teilen von Zügen, also die getrennte Betrachtung von Amazonenbewegungen und Schussaktionen, die Effizienz des Algorithmus erhöht werden. Diese und andere Verbesserungen, wie das Pruning von sinnlosen Zügen, haben sich als vorteilhaft in der Anwendung von MCTS auf Amazons erwiesen [1].

Optimierung der MC-Simulation

In dieser Arbeit wurde für das Durchführen der Simulationen keine zufälligen Züge ausgewählt. Um wesentlich mehr Zeit zu ersparen, wurde dabei immer nur eine zufällig ausgewählte Amazone betrachtet. Für jeden Zug in einer Simulation wird eine zufällige Zahl von 1-4 gewählt. Diese Zahl sagt aus für welche Amazone die möglichen Züge für die Simulation berechnet werden soll (es werden nur Amazonen betrachtet, die auch noch mögliche Züge haben). Dadurch reduziert sich die Anzahl der möglichen Züge erheblich und dessen Berechnungen, die durchgeführt werden müssten. Denn es wurde für jeden Zug innerhalb der Simulation die Bewertungsfunktion hinzugezogen, sodass nur gute Züge innerhalb der Simulation ausgeführt werden. Auch in diesem Kapitel 5.1 wurde der MCTS mit rein zufälligen Zügen getestet.

Recycling von Zweigen

Eine weitere mögliche Verbesserung des MCTS wäre das **Recycling von Zweigen**, anstatt diese bei jedem Zug neu berechnen zu müssen. In der aktuellen Implementierung werden nach jedem Zug alle bereits simulierten Knoten verworfen und der Baum wird von Grund auf neu aufgebaut. Dies könnte man umsetzen indem der existierenden Baum beibehalten wird, sodass der Unterbaum des zuletzt gewählten Zuges als neuer Suchbaum genutzt wird. Dieses **Baumrecycling** oder die Verwendung einer **Transpositionstabelle**, reduziert die Anzahl der benötigten Berechnungen, da bereits durchgeführte Simulationen und Berechnungen wiederverwendet werden können. Dies ist besonders in späteren Spielphasen von Vorteil, wenn der Baum bereits umfangreiche Informationen enthält, die wiederverwendet werden können.
Hinweis: Diese Verbesserung wurde in dieser Arbeit jedoch nicht implementiert und wurde hier nur als mögliche Verbesserung vorgestellt.

Erweiterung der Wurzel um Top 30 Züge

Eine weitere vielversprechende Verbesserung besteht darin, bei der Erweiterung der Wurzel nur die besten Züge zu berücksichtigen. In Anlehnung an den Ansatz von Lorentz in dessen Arbeit [6] wird die Wurzel zu aller erst um eine bestimmte Anzahl von Top Zügen erweitert, die basierend auf der Bewertungsfunktion als die vielversprechendsten ermittelt wurden. Anstatt, dass die Wurzel also um x zufällige Knoten erweitert wird, selektiert dieser Ansatz die 30 besten Züge, um die Suche effizienter zu gestalten, indem er sich auf die besten Spielzüge konzentriert und dadurch unnötige Berechnungen für weniger erfolgversprechende Züge vermeidet und nur innerhalb diesen 30 Zügen entscheidet. Das erhöht im Allgemeinen die Spielstärke und vermeidet das möglicherweise durch die stochastische Natur von MCTS nur wenig vielversprechende Kindknoten an die Wurzel erweitert werden.

3.5 Bewertungsfunktion

In dieser Arbeit wird für alle behandelten Algorithmen – einschließlich *Alpha-Beta-Pruning*, *Monte Carlo Tree Search* (MCTS), *MTD(f)* und *Principal Variation Search* (PVS) – dieselbe Bewertungsfunktion verwendet. Diese Funktion ermöglicht eine konsistente Bewertung von Spielsituationen und stellt eine vergleichbare Grundlage für die Performance der verschiedenen Suchalgorithmen dar.

Der Ansatz für die Bewertungsfunktion orientiert sich stark an der Arbeit von *Lieberum* [5], welcher eine detaillierte Analyse der Bewertung von Spielsituationen im *Amazons*-Spiel durchführte. Diese Bewertungsfunktion nutzt zwei Hauptkriterien:

- **Territoriale Kontrolle**: Basierend auf den minimalen Distanzen der Spielfelder zu den Amazonen jedes Spielers.
- **Mobilität**: Basierend auf der Anzahl der möglichen Züge für die Amazonen jedes Spielers.

3.5.1 Territoriale und Positionale Bewertung

Das Ziel von Amazons besteht darin, am Ende mehr leere Felder zu kontrollieren als der Gegner. Diese Kontrolle wird durch die Distanz der Amazonen zu den freien Feldern gemessen. Dabei wer-

den zwei unterschiedliche Distanzmetriken verwendet, wie von Lieberum in [5] beschrieben: die Königin-Distanz $d_1(a, b)$ (Anzahl der benötigten Züge in Form der Schachdame) und die Königs-Distanz $d_2(a, b)$ (Anzahl der benötigten Züge in Form des Schachkönigs) zwischen zwei Feldern a und b. Diese Distanzen geben an, wie viele Züge eine Amazone benötigt, um ein bestimmtes Feld zu erreichen, wobei $d_1(a, b)$ die minimalen Züge für eine Amazone und $d_2(a, b)$ die minimalen Züge für einen König beschreibt.

Die territoriale Kontrolle eines Spielers j wird durch die Distanzfunktion $D_i^j(a)$ beschrieben, die die minimalen Distanzen von einer Amazone des Spielers j zu einem Feld a angibt:

$$D_i^j(a) = \min\{d_i(a, b) \mid \text{b ist von einer Amazone des Spielers j besetzt}\}$$

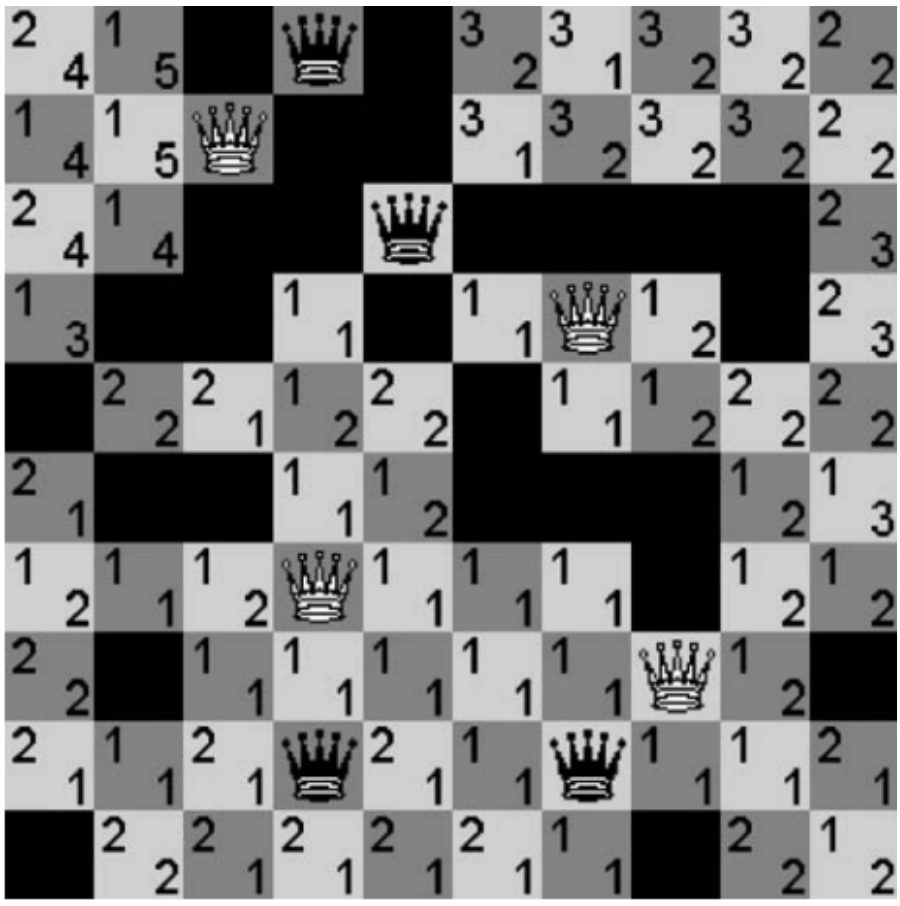

Abbildung 3.3: Berechnung der Minimalen Distanzen $D_i^j(a)$ [5]

Abbildung 3.3 illustriert ein Beispiel für die Berechnung der minimalen Distanzen $D_1^j(a)$ und $D_2^j(a)$. Dabei enthält die obere linke Ecke eines Feldes den Wert $D_1^j(a)$ und die untere rechte Ecke den Wert $D_2^j(a)$.

Auf Basis dieser Distanzen erfolgt die Berechnung der globalen territorialen Bewertungen t_1 und t_2 und lassen sich wie folgt definieren:

$$t_i = \sum_{\text{leere Felder } a} \Delta(D_i^1(a), D_i^2(a))$$

Demnach der Spieler einen Vorteil, dessen Amazonen eine geringere Distanz zu einem bestimmten Feld aufweisen, als die des Gegners. Die Funktion $\Delta(n, m)$ definiert genau dies und sieht dabei wie folgt aus:

$$\Delta(n,m) = \begin{cases} 0, & \text{wenn } n = m = \infty, \\ \kappa, & \text{wenn } n = m < \infty, \\ 1, & \text{wenn } n < m, \\ -1, & \text{wenn } n > m, \end{cases}$$

wobei κ eine Konstante ist, die den Vorteil des Startspielers approximiert. Der Wert κ wurde in Lieberums Arbeit [5] auf etwa $\frac{1}{5}$ gesetzt, um Stabilität in der Bewertung sicherzustellen.
Aus Sicht des Spieler 1: Eine 1 *bedeutet dabei, dass der Vorteil um* 1 *erhöht wird und eine* -1*, dass dieser verkleinert wird etc.*

Die lokale Bewertung der Felder wird durch eine gewichtete Summe der Distanzen ausgedrückt und erfolgt durch die Berechnung von c_1 und c_2. Die Formel für die Königin-Distanzen c_1 lautet:

$$c_1 = 2 \sum_{\text{leere Felder } a} 2^{-D_1^1(a)} - 2^{-D_1^2(a)}$$

Diese Formel belohnt den Spieler, der näher an einem Feld ist, basierend auf den Distanzen. Ähnlich berücksichtigt c_2 die Königsdistanzen:

$$c_2 = \sum_{\text{leere Felder } a} \min(1, \max(-1, \frac{D_2^2(a) - D_2^1(a)}{6}))$$

Die territoriale Gesamtbewertung t kombiniert die verschiedenen Aspekte der territorialen Bewertung, wobei dynamische Gewichtungen verwendet werden, um den Einfluss der verschiedenen Faktoren im Verlauf des Spiels (genauer der Spielphasen) anzupassen. Die territoriale Gesamtbewertung wird dann als gewichtete Summe der Einzelbewertungen ausgedrückt:

$$t = f_1(x)t_1 + f_2(x)c_1 + f_3(x)c_2 + f_4(x)t_2$$

wobei die Funktionen $f_i(x)$ für eine schrittweise Anpassung der Gewichtungen im Verlauf des Spiels sorgen. Es wurde eine angepasste Version dieser Gewichtungsfunktionen integriert, um die Gewichtung basierend auf dem Spielverlauf dynamisch zu gestalten. Dabei sollte jedoch stets $\sum_i f_i(x) = 1$ sein. Die Definitionen von f_i mit $i \in \{1, 2, 3, 4\}$ wurden aus der Arbeit von Yanik [15] entnommen. Dabei wurde $v = 0.7$ gewählt und x als der Spielfortschritt bezeichnet, welcher besagt wie viele Felder des Feldes bereits belegt sind:

$$x = \frac{Anzahl\_belegter\_Felder}{100}$$

Die Definitionen der einzelnen $f_i(x)$ sehen wie folgt aus:

$$
\begin{aligned}
f_1(x) &= v \cdot x \\
f_2(x) &= (1 - v) \cdot x \\
f_3(x) &= v \cdot (1 - x) \\
f_4(x) &= (1 - v) \cdot (1 - x)
\end{aligned}
$$

Dabei soll $f_1(x)$ so aussehen, dasss diese mit der Zeit immer stärker an Gewichtung bekommt, da die Bewertung t_1 sehr gute Ergebnisse nach der Auffüllphase des Spiels liefert. Das genaue Gegenteil von t_1 stellt hier t_2 dar. Diese Bewertung belohnt eine ausgewogene Verteilung der Amazonen eines Spielers, sodass der andere Spieler nicht eine solche Verteilung erreicht. Die Kontrolle über das Zentrum des Spielfelds soll hier gewährleistet werden und spielt eine besonders wichtige Rolle am Anfang des Spiels. Die Werte c_1 und c_2 verfeinern die Bewertung, dadurch dass sie von der Qualität der lokalen Vorteile abhängen. Es soll in der Bewertung der Position ein guter und fließender Übergang zwischen den Phasen des Spiels entstehen [5].

3.5.2 Mobilitätsbewertung

Die Mobilität der Amazonen ist ein weiterer wesentlich wichtiger Aspekt bei der Evaluierung einer Spielposition in Amazons, genannt Mobilitätsbewertung. Sie wird definiert als die Anzahl der Felder, die die jeweiligen Amazonen innerhalb der Bewegungsreichweite erreichen können. Dies ist besonders in den frühen Phasen des Spiels wichtig, wenn Amazonen von dem Gegenspieler eingekreist werden können, was ihre Bewegungsfreiheit stark einschränkt.

Für jede Amazone wird die Anzahl der Felder berechnet, die sie unter Berücksichtigung der Blockaden durch andere Amazonen oder Pfeile erreichen kann. Sprich es wird geprüft, wie viele freie Felder in allen acht Bewegungsrichtungen einer Amazone erreichbar sind.

Die Mobilitätsevaluierung für einen Spieler j erfolgt dann durch die Summierung der Mobilität aller Amazonen des Spielers:

$$
m_j = \sum_{\text{Amazonen von Spieler j}} \text{Mobility}(a)
$$

Schließlich wird die Mobilitätsdifferenz zwischen den Spielern berechnet und zur Gesamtbewertung hinzugefügt:

$$
m = m_1 - m_2
$$

3.5.3 Gesamtevaluation

Die Gesamtevaluation kombiniert die territoriale und die Mobilitätsbewertung und lautet:

$$\text{Evaluation} = t + m$$

Dieser Ansatz ermöglicht eine ganzheitliche Bewertung der Spielsituation, bei der sowohl die territoriale und positionale Kontrolle als auch die Bewegungsfreiheit der Amazonen berücksichtigt werden. Bei der Berechnung der Evaluation zeichnet sich ein Vorteil für den Spieler 1 durch positive Werte der Evaluation aus ($t + m > 0$) aus, währenddessen für den Spieler 2 negative Werte einen Vorteil bedeuten ($t + m < 0$).

4 Durchführung der Tests

Um die Leistungsfähigkeit der Suchalgorithmen *Alpha-Beta*, *Principal Variation Search (PVS)*, *MTD(f)* und *Monte Carlo Tree Search (MCTS)* im Kontext des Spiels *Amazons* systematisch zu vergleichen, wurden eine Reihe von Tests durchgeführt. Diese Tests zielen darauf ab, die Algorithmen unter realistischen Spielbedingungen zu evaluieren und dabei zentrale Aspekte wie Rechenzeit, Zugqualität, Speichereffektivität und Siegquote zu messen.

4.1 Testumgebung

Die Implementierung der Algorithmen erfolgte in der Programmierumgebung Visual Studio Code und der Programmiersprache Python. Um eine faire Vergleichbarkeit der Ergebnisse sicherzustellen, liefen alle Algorithmen auf derselben Hardware-Konfiguration. Damit die Tests unter gleichen Bedingungen stattfinden und konsistente Ergebnisse gewährleistet werden können, wurden alle Tests auf einem Rechner mit folgenden Spezifikationen durchgeführt:

- **CPU**: AMD Ryzen 9 5900X, 12 Kerne, 3.7 GHz - 4.4 GHz
- **Arbeitsspeicher**: 32 GB DDR4
- **Betriebssystem**: Windows 10

Verwendete Software:

- **Programmiersprache**: Python 3.11.4
- **Bibliotheken**: Numpy 1.23.5
- **Entwicklungsumgebung**: Visual Studio Code 1.94.2

4.2 Testaufbau

Um einen ausgewogenen Vergleich der Algorithmen durchzuführen, wurde unter anderem eine Serie von Spielen zwischen den Algorithmen angesetzt. Dabei traten die Algorithmen bei bestimmten Tests in einem Round-Robin-Format gegeneinander an, d.h., jeder Algorithmus spielte eine Reihe von Partien gegen jeden anderen Algorithmus. Jeder Algorithmus wurde sowohl in der Rolle des ersten (Spieler Weiß) als auch des zweiten Spielers (Spieler Schwarz) getestet, um mögliche Vorteile durch die Startposition zu eliminieren. Für andere Tests, wie der Zugqualität, Rechenzeit und Speichereffektivität wurden die Algorithmen in einer präparierten Spielposition getestet. Für jeden Test wurden insgesamt *50* Durchgänge durchgeführt (wenn nichts explizit über die Anzahl der Tests angegeben wurde), um statistisch belastbare Ergebnisse zu gewährleisten.

4.3 Testmetriken

Zur Bewertung der Algorithmen wurden verschiedene Metriken herangezogen, die unterschiedliche Aspekte der Spielstärke und Effizienz der Algorithmen abdecken:

- **Iterationsanzahl**[1]: Dabei wird gezählt wie oft ein Algorithmus Knoten, während des Berechnungsprozesses eines Zuges, untersuchte. Dadurch gibt es einen Einblick in die Effizienz der Suchalgorithmen im Umgang mit der Komplexität des Spielbaums.
- **Rechenzeit**: Die durchschnittliche Zeit, die jeder Algorithmus pro Zug benötigte, wurde aufgezeichnet, um die Effizienz der Algorithmen zu bewerten.
- **Zugqualität**[1,2]: Die Qualität der Züge wurde anhand der Bewertungsfunktion des Spiels gemessen. Hierbei wurde erfasst, wie oft ein Algorithmus den bestmöglichen Zug im Vergleich zu anderen Algorithmen wählte.
- **Speichereffizienz**[2]: Dabei wurde die Größe der jeweiligen Transpositionstabelle gemessen. Diese Messung gibt Aufschluss darüber, wie viel Speicherplatz jeder Algorithmus benötigt, um bereits besuchte Zustände effizient wiederzuverwenden, insbesondere in komplexeren Spielphasen.
- **Siegquote**[1]: Die Anzahl der gewonnenen Spiele pro Algorithmus wurde gemessen, um die allgemeine Spielstärke zu evaluieren.

[1] *hierbei wurden die Tests sowohl in einem Zeitlimit von 30 Sekunden als auch 60 Sekunden getestet.*
[2] *hierbei wurden die Tests in 3 unterschiedlichen Phasen des Spiels getestet.*

4.4 Testablauf

Für jedes Spiel im Round-Robin-Format wurden die folgenden Schritte durchlaufen:

1. **Initialisierung des Spiels**: Das Brett wird in der Anfangsstartaufstellung initialisiert, und die Spieler (Algorithmen) werden zugewiesen.
2. **Zugberechnung**: Die Algorithmen berechnen abwechselnd ihren nächsten Zug. Dabei führen die Algorithmen eine Suche des besten Zuges durch, bis die vorgegebene Zeit von 30 oder 60 Sekunden erreicht wurde.
3. **Auswertung der Züge**: Jeder berechnete Zug wird sofort ausgewertet und auf dem Spielbrett ausgeführt.
4. **Statistikaufzeichnung**: Während des Spiels werden relevante Statistiken wie Zugqualität und Iterationsanzahl aufgezeichnet.
5. **Ende des Spiels**: Sobald ein Spieler keinen gültigen Zug mehr ausführen kann, endet das Spiel und der Gewinner wird bestimmt.
6. **Ergebnisaufzeichnung**: Die Ergebnisse des Spiels (z.B. Sieger, Zuganzahl, weitere Metriken) werden in einer JSON-Datei gespeichert, um die spätere Analyse zu erleichtern.

Für die übrigen Tests (in denen die Algorithmen nicht direkt gegeneinander spielten) wurden statt der Standardanfangsstellung vorab gespielte Spielpositionen als Ausgangsaufstellung verwendet. Dort wurden währenddessen andere Metriken des Test aufgezeichnet und ebenfalls in einer JSON-Datei gespeichert.

4.5 Herausforderungen und Optimierungen

Während der Tests traten einige spezifische Herausforderungen auf, die berücksichtigt wurden:

- **Zufälligkeit bei MCTS**: Da MCTS auf einer stochastischen Suche basiert, wurden die Tests mehrfach mit unterschiedlichen Zufallsseed-Werten durchgeführt, um sicherzustellen, dass die Ergebnisse repräsentativ sind.
- **Vergleich der Laufzeiten für bestimmte Tiefen**: Dadurch das MCTS nicht wie Alpha-Beta mit Tiefen direkt arbeitet, wurde für den MCTS die Limitierung der Tiefe wie folgt durchgeführt: Für den MCTS-Algorithmus wurde die Tiefe als Baumebene definiert, wobei die Wurzelknoten die Tiefe 0 repräsentieren. Jeder nachfolgende Knoten, der ein direkter Nachkomme eines Knotens der vorherigen Ebene ist, zählt als eine tiefere Ebene im Baum. Zum Beispiel hätte ein Kindknoten der Wurzel die Tiefe 1, und ein weiterer Kindknoten von diesem Knoten würde die Tiefe 2 aufweisen.[1]
- **Suchtiefen-Anpassung**: Die Tiefe der Suchen für Alpha-Beta, PVS und MTD(f) wurde so gewählt, dass die Berechnungszeit im Rahmen blieb, jedoch genug Tiefe erreicht wurde, um aussagekräftige Züge zu berechnen.
- **Berechnung der durchschnittlichen Iterationszahlen**: Da die Spiele im Round-Robin-Format unterschiedlich verlaufen und zu unterschiedlichen Zeitpunkten enden und dadurch die Iterationszahlen beim MCTS gegen Ende des Spiels drastisch ansteigen, während sie bei Alpha-Beta, PVS und MTD-f stark abnehmen, konnten die jeweils letzten berechneten Iterationszahlen nicht berücksichtigt werden, da sie als starke Ausreißer gelten.
- **Lange Berechnungszeit der Evaluierungsfunktion**: Die Evaluierungsfunktion benötigt relativ lange für die Berechnung der Spielposition, wodurch die Algorithmen nicht in allzu großen Tiefen getestet werden konnten. Daher basiert der Vergleich der Laufzeiten in Test 5.3 auf einem bereits fortgeschrittenen Spielzustand, da die Berechnung des besten Zugs in größeren Tiefen in der Anfangsphase von Amazons zu viel Zeit in Anspruch genommen hätte.

Die Tests wurden durchgeführt, um eine fundierte Datenbasis zu schaffen, die es ermöglicht, die Leistungsfähigkeit und Effizienz der Algorithmen in verschiedenen Spielsituationen zu vergleichen. Die gesammelten Daten bilden die Grundlage für die Analyse und Bewertung der Algorithmen, die im folgenden Kapitel detailliert vorgestellt wird.

[1] Dadurch konnte MCTS zu diesem Testzweck miteinbezogen werden, jedoch stellt sich die Frage, ob dies wirklich sinnvoll und aussagekräftig ist.

5 Ergebnisse

5.1 Ermittlung der besten MCTS-Version

Das Ziel dieser Untersuchung bestand zunächst darin, aus einer Vielzahl von Monte Carlo Tree Search (MCTS)-Varianten die bestmögliche Konfiguration für den Einsatz im Spiel Amazons zu identifizieren. Hierbei wurden unterschiedliche Simulationstiefen, Zugpriorisierungen und Werte für den Explorationsparameter getestet, um herauszufinden, welche Einstellungen MCTS im Vergleich zur Alpha-Beta-Suche die beste Leistung bringen. Die folgende Analyse beschreibt die verschiedenen MCTS-Versionen und die erzielten Ergebnisse im Vergleich zu Alpha-Beta.

Die MCTS-Varianten wurden mit einem Zeitlimit von 60 Sekunden und 30 Sekunden pro Zug getestet. Die Ergebnisse dieser Tests, die die Siegquote der jeweiligen MCTS-Version gegen Alpha-Beta dokumentieren, sind in den Tabellen 5.2 und 5.3 zusammengefasst. Die Bezeichnungen MCTS (v.1), MCTS (v.2), MCTS (v.3), usw. repräsentieren die unterschiedlichen Konfigurationen der MCTS-Parameter und sehen wie folgt aus:

Tabelle 5.1: MCTS-Varianten mit den jeweiligen Konfigurationen

MCTS-Version	**Konfiguration**
MCTS (v.1.1)	Simulationstiefe von 6
MCTS (v.1.2)	Simulationstiefe von 10
MCTS (v.2.1)	Simulationstiefe von 4 und Zugpriorisierung bei den Playouts
MCTS (v.2.2)	Simulationstiefe von 6 und Zugpriorisierung bei den Playouts
MCTS (v.2.3)	Simulationstiefe von 8 und Zugpriorisierung bei den Playouts
MCTS (v.2.4)	Simulationstiefe von 10 und Zugpriorisierung bei den Playouts
MCTS (v.3.1)	Einstellungen des MCTS (v.2.2) und Explorationswert auf 0.7 geändert
MCTS (v.3.2)	Einstellungen des MCTS (v.2.2) und Explorationswert auf 1.0 geändert
MCTS (v.3.3)	Einstellungen des MCTS (v.2.2) und Explorationswert auf 1.7 geändert

Anmerkung: Bei den Versionen **MCTS (v.1.1)** bis **MCTS (v.2.4)** wurde der **Explorationswert konstant auf 1.4** eingestellt, während bei den Versionen **MCTS (v.3.1)**, **MCTS (v.3.2)** und **MCTS (v.3.3)** der Explorationswert auf 0.7, 1.0 bzw. 1.7 angepasst wurde.

5.1.1 Ergebnisse mit 60-Sekunden Zeitlimit

In Tabelle 5.2 sind die Spielergebnisse für die Spiele mit einem Zeitlimit von 60 Sekunden dargestellt. Für die Tests wurden jeweils 10 Spiele als Spieler Weiß und 10 Spiele als Spieler Schwarz durchgeführt.

Algorithmus 1	**Algorithmus 2**	**Siegquote Algorithmus 1**
Alpha-Beta	MCTS (v.1.1)	95%
Alpha-Beta	MCTS (v.1.2)	100%
Alpha-Beta	MCTS (v.2.1)	95%
Alpha-Beta	MCTS (v.2.2)	90%
Alpha-Beta	MCTS (v.2.3)	90%
Alpha-Beta	MCTS (v.2.4)	95%
Alpha-Beta	MCTS (v.3.1)	95%
Alpha-Beta	MCTS (v.3.2)	95%
Alpha-Beta	MCTS (v.3.3)	90%

Tabelle 5.2: Spielergebnisse MCTS Versionen vs Alpha Beta (60 Sekunden Zeitlimit)

5.1.2 Ergebnisse mit 30-Sekunden Zeitlimit

In Tabelle 5.3 sind die Spielergebnisse für die Spiele mit einem Zeitlimit von 30 Sekunden zusammengefasst. Auch hier wurden für die Tests wurden jeweils 10 Spiele als Spieler Weiß und 10 Spiele als Spieler Schwarz durchgeführt.

Algorithmus 1	Algorithmus 2	Siegquote Algorithmus 1
Alpha-Beta	MCTS (v.1.1)	90%
Alpha-Beta	MCTS (v.1.2)	100%
Alpha-Beta	MCTS (v.2.1)	90%
Alpha-Beta	MCTS (v.2.2)	80%
Alpha-Beta	MCTS (v.2.3)	90%
Alpha-Beta	MCTS (v.2.4)	100%
Alpha-Beta	MCTS (v.3.1)	85%
Alpha-Beta	MCTS (v.3.2)	100%
Alpha-Beta	MCTS (v.3.3)	100%

Tabelle 5.3: Spielergebnisse MCTS Versionen vs Alpha Beta (30 Sekunden Zeitlimit)

5.1.3 Auswahl der besten MCTS-Version

Die Auswahl der besten MCTS-Version ist besonders wichtig für das Durchführen der weiteren Tests. Demnach schneidet die Version **MCTS (v.2.2)** basierend auf den oben gezeigten Testergebnissen am besten ab. Diese Version erreichte eine Siegquote des Alpha-Beta-Gegners von **90%** im 60-Sekunden-Zeitlimit und **80%** im 30-Sekunden-Zeitlimit. Die geringere Siegquote des Alpha-Beta-Algorithmus zeigt, dass **MCTS (v.2.2)** die stärkste Leistung gegen Alpha-Beta erreichen konnte.

Daher wird **MCTS (v.2.2)** als die beste Version ausgewählt und in den folgenden Tests weiter untersucht.

5.2 Vergleich der Iterationen pro Zug

Die Anzahl der durchgeführten Iterationen pro Zug wurde für jeden Algorithmus gemessen, um die Effizienz der Algorithmen in Bezug auf die Berechnungsgeschwindigkeit zu bewerten.

Die Anzahl der Iterationen pro Zug wurde für die Algorithmen, Monte Carlo Tree Search (MCTS), Alpha-Beta, PVS und MTD(f) in zwei Szenarien gemessen: einmal mit einem Zeitlimit von 60 Sekunden und einmal mit einem Zeitlimit von 30 Sekunden. Die Ergebnisse sind in den folgenden Grafiken 5.1 und 5.2 dargestellt.

5.2.1 Iterationsergebnisse mit 60-Sekunden-Zeitlimit

In Abbildung 5.1 werden die durchschnittlichen Iterationen pro Zug der jeweiligen Algorithmen im ersten Spiel mit einem Zeitlimit von 60 Sekunden pro Zug aufgeführt.

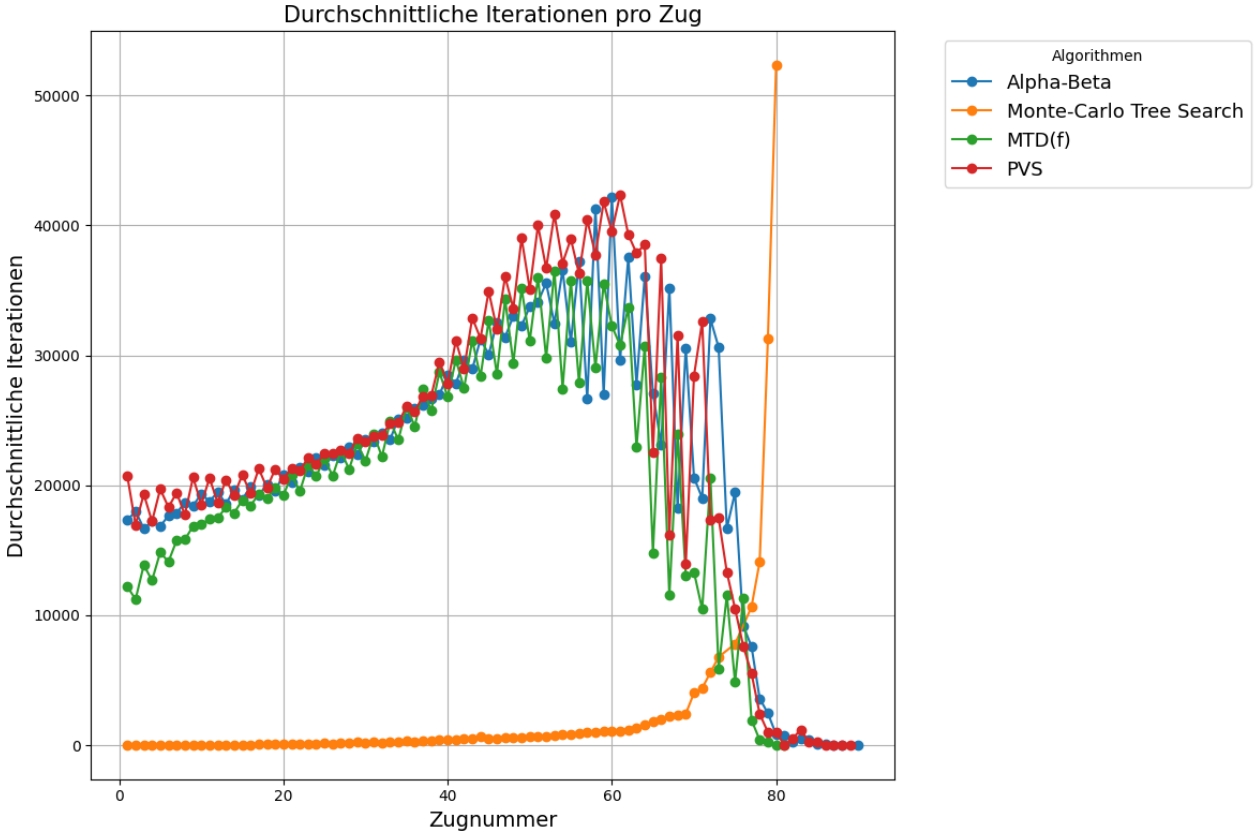

Abbildung 5.1: Vergleich der Iterationen pro Zug bei Zeitlimit von 60 Sekunden zwischen Alpha-Beta, MCTS, MTD(f) und PVS

5.2.2 Iterationsergebnisse mit 30-Sekunden-Zeitlimit

In Abbildung 5.2 sind die Iterationen mit einem Zeitlimit von 30 Sekunden dargestellt.

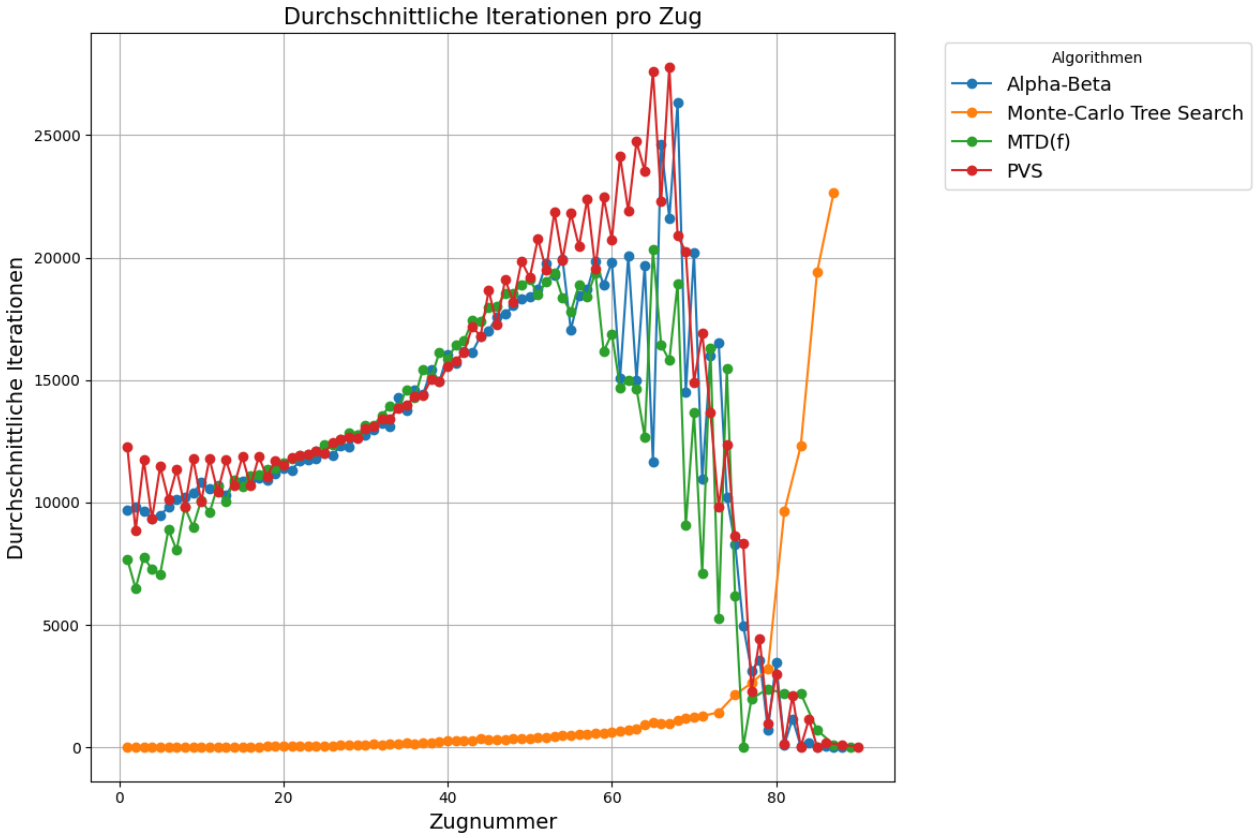

Abbildung 5.2: Vergleich der Iterationen pro Zug bei Zeitlimit von 30 Sekunden zwischen Alpha-Beta, MCTS, MTD(f) und PVS

5.3 Vergleich der Laufzeiten

Die Laufzeit wurde ebenfalls für jeden Algorithmus gemessen. Dabei wird die Durchschnittliche Laufzeit pro Zug (in Sekunden) gemessen, wie lange die Algorithmen für die jeweilige Tiefe benötigt haben. Durch die im Abschnitt 4.5 erwähnten Herausforderungen bezüglich der langen Berechnungszeit der Evaluierungsfunktion wurde dieser Test auf einem bereits fortgeschrittenen Spielfeld (aus einem Spiel zwischen MCTS und Alpha-Beta nach 40 Zügen) durchgeführt.

Hinweis: Für den MCTS-Algorithmus wurde die Tiefe als Baumebene definiert, wobei die Wurzelknoten die Tiefe 0 repräsentieren. Jeder nachfolgende Knoten, der ein direkter Nachkomme eines Knotens der vorherigen Ebene ist, zählt als eine tiefere Ebene im Baum. Zum Beispiel hätte ein Kindknoten der Wurzel die Tiefe 1, und ein weiterer Kindknoten von diesem Knoten würde die Tiefe 2 aufweisen.

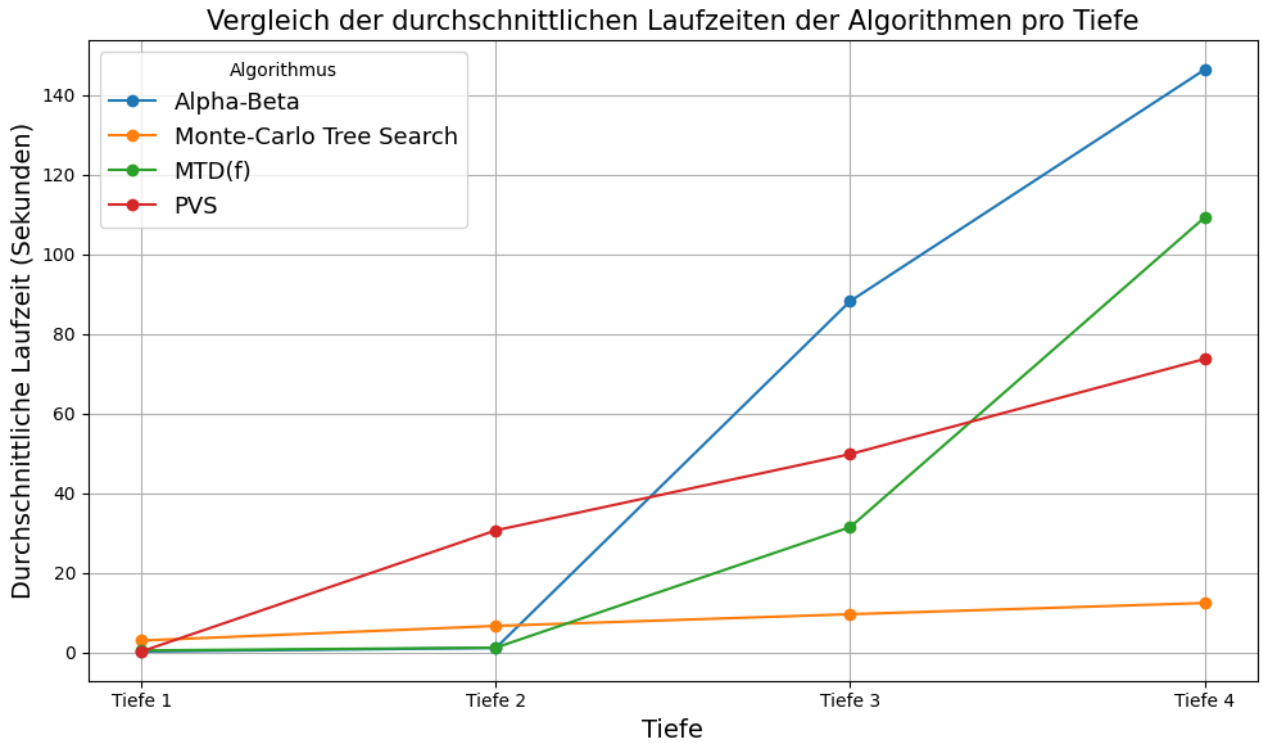

Abbildung 5.3: Vergleich der durchschnittlichen Laufzeiten der Algorithmen pro Tiefe

Algorithmus	**Tiefe 1**	**Tiefe 2**	**Tiefe 3**	**Tiefe 4**
Alpha-Beta	0,2669	1.1518	88.2807	146.4323
PVS	0,2751	30.7017	49.8422	73.7535
MTD(f)	0,5230	1.2275	31.5056	109.2936
MCTS	3.0083	6.6916	9.6342	12.4464

Tabelle 5.4: Vergleich der durchschnittlichen Laufzeiten der Algorithmen pro Tiefe

5.4 Effizienz in verschiedenen Spielphasen

Um die Effizienz der Algorithmen in unterschiedlichen Spielphasen zu bewerten, wurden Metriken wie die Anzahl der Iterationen und die relative Bewertung der Züge analysiert. Die Bewertung der Effizienz wurde in drei Phasen des Spiels durchgeführt: der frühen, mittleren und Endspielphase.

Die Spielphasen wurden wie folgt definiert:

- **Frühe Spielphase:** Die Algorithmen wurden nach **10 Zügen** auf dem Spielbrett ausgeführt.
- **Mittlere Spielphase:** Die Algorithmen wurden nach **30 Zügen** auf dem Spielbrett ausgeführt.
- **Endspielphase:** Die Algorithmen wurden nach **50 Zügen** auf dem Spielbrett ausgeführt.

Für jede Spielphase beträgt die Anzahl der durchgeführten Tests **20 Durchläufe**. Die angegebenen Werte in den Tabellen stellen den Durchschnitt der Ergebnisse aus diesen 20 Durchläufen dar. Die Algorithmen hatten jeweils 60 Sekunden Zeit, um den nächsten Zug als Spieler 1 zu berechnen. Eine höhere Bewertung des berechneten Zuges spiegelt dabei die Qualität der Suche wider.

Die Ergebnisse dieser Analyse, einschließlich der Iterationsanzahl und der Bewertung des besten Zugs, sind in den Tabellen 5.5, 5.6 und 5.7 für die frühen, mittleren und Endspielphasen zusammengefasst.

5.4.1 Frühe Spielphase

Das für die Algorithmen getestete Spielbrett weist eine anfängliche Bewertung von $7{,}53$ auf. Spieler Weiß hat demnach einen kleinen Vorteil.

Algorithmus	**Iterationen**	**Bewertung des besten Zugs**
Alpha-Beta	21585.10	17.55
MCTS	38.80	15.16
MTD(f)	21762.20	16.11
PVS	22193.20	17.55

Tabelle 5.5: Vergleich der Iterationen und Zugbewertung in der frühen Spielphase

5.4.2 Mittlere Spielphase

Das für diesen Test verwendete Spielbrett hat eine Anfangsbewertung von $-1{,}20$. Spieler Schwarz ist leicht im Vorteil.

Algorithmus	**Iterationen**	**Bewertung des besten Zugs**
Alpha-Beta	26431.40	2.16
MCTS	170.55	5.66
MTD(f)	27967.65	5.66
PVS	25635.95	5.66

Tabelle 5.6: Vergleich der Iterationen und Zugbewertung in der mittleren Spielphase

5.4.3 Endspielphase

Das Spielbrett für die Endspielphase besitzt eine anfängliche Evaluierung von $26{,}06$. Spieler Weiß ist klar im Vorteil.

Algorithmus	Iterationen	Bewertung des besten Zugs
Alpha-Beta	38047.85	27.36
MCTS	695.55	26.78
MTD(f)	38400.30	24.50
PVS	52061.35	23.96

Tabelle 5.7: Vergleich der Iterationen und Zugbewertung in der Endspielphase

5.5 Speicherbedarf der Transpositionstabelle

Zur Bewertung der Effizienz der Transpositionstabelle wurde der Speicherbedarf analysiert. Die Ergebnisse sind in Tabelle 5.8 zusammengefasst. Auch hier wurde wieder in den unterschiedlichen Spielphasen differenziert wie in 5.4.

	Speicherbedarf (Kilobytes)		
Algorithmen	**Frühe Spielphase**	**Mittlere Spielphase**	**Endspielphase**
Alpha-Beta	3,328	13,056	13,056
PVS	0,832	1,584	207,616
MTD(f)	26,032	13,056	207,616

Tabelle 5.8: Speicherbedarf der Transpositionstabelle

5.6 Siegquote der Algorithmen

Bei der Ermittlung der Siegquote sind alle vier Algorithmen im Round-Robin-Format gegeinander angetreten, jeweils im 60 und 30 Sekunden Zeitlimit. Die Siegquote für das jeweilige Zeitlimit j mit $j \in \{60, 30\}$ errechnet sich dann aus

$$\frac{Anzahl\_der\_gewonnenen\_Spiele_j}{Gesamtanzahl\_der\_gespielten\_Spiele_j}$$

Dadurch ergibt sich für jeden Algorithmus eine prozentuale Angabe seiner erzielten Siegquote im vergleich zu den anderen Algorithmen. Die allgemeinen Siegquoten werden in den Tabellen 5.10 und 5.12 zusammengefasst, währenddessen die Tabellen 5.9 und 5.11 eine Matrix der jeweiligen Siegquote zwischen den Algorithmen darstellen.

5.6.1 Ergebnisse des 60-Sekunden Zeitlimits

Algorithmus	Alpha-Beta	MCTS	PVS	MTD(f)
Alpha-Beta	-	92%	94%	96,67%
MCTS	8%	-	12%	50%
PVS	6%	88%	-	100%
MTD(f)	3,33%	50%	0%	-

Tabelle 5.9: Siegquoten der Algorithmen in Prozent: Die Zelle (i, j) zeigt die Siegquote von Algorithmus i gegen Algorithmus j - (Zeile i und Spalte j)

Algorithmus	Siegquote
Alpha-Beta	93,85%
PVS	59,23%
MTD(f)	17,78%
MCTS	19,23%

Tabelle 5.10: Siegquoten der verschiedenen Algorithmen bei einem Zeitlimit von 60 Sekunden

5.6.2 Ergebnisse des 30-Sekunden Zeitlimits

Algorithmus	Alpha-Beta	MCTS	PVS	MTD(f)
Alpha-Beta	-	94%	93,48%	100%
MCTS	6%	-	10,20%	50%
PVS	6,52%	89,80%	-	100%
MTD(f)	0%	50%	0%	-

Tabelle 5.11: Siegquoten der Algorithmen in Prozent: Die Zelle (i, j) zeigt die Siegquote von Algorithmus i gegen Algorithmus j - (Zeile i und Spalte j)

Algorithmus	Siegquote
Alpha-Beta	94,62%
PVS	56,92%
MTD(f)	16,67%
MCTS	21,54%

Tabelle 5.12: Siegquoten der verschiedenen Algorithmen bei einem Zeitlimit von 30 Sekunden

5.7 Asymmetrischer Test: MCTS vs Alpha-Beta

In den bisherigen Experimenten dieser Arbeit schnitt der Monte Carlo Tree Search (MCTS) Algorithmus im direkten Vergleich mit dem Alpha-Beta-Algorithmus signifikant schlechter ab. Um die Leistungsfähigkeit des MCTS in einem Szenario zu untersuchen, in dem ihm zusätzliche Ressourcen zur Verfügung stehen, wird in dieser Sektion ein asymmetrischer Test durchgeführt. Ziel ist es zu ermitteln, ob und unter welchen Bedingungen MCTS seine Gewinnquote gegenüber Alpha-Beta erhöhen kann.

Dafür wurden unterschiedliche Zeitlimits für die jeweiligen Algorithmen getestet, jedoch stets ein höheres Zeitbudget für den MCTS. Durch die schrittweise Anpassung der Berechnungszeit von Alpha-Beta und MCTS soll die Schwelle gefunden werden, ab wann der MCTS konkurrenzfähig wird. Die Ergebnisse sagen darüber aus, wie stark MCTS von einer erhöhten Berechnungszeit profitiert und wie viel zusätzliche Zeit erforderlich ist, um gegen Alpha-Beta anzutreten.

5.7.1 Ergebnisse

Zeitlimit [in Sekunden]		
Alpha-Beta	**MCTS**	**Siegquote MCTS**
10	60	20%
20	60	15%
30	60	10%
40	60	0%
30	90	5%
30	120	15%
10	90	25%

Tabelle 5.13: Siegquote des MCTS gegen Alpha-Beta bei asymmetrischen Bedingungen

6 Diskussion

In diesem Kapitel werden die Ergebnisse der durchgeführten Tests im Kontext der Algorithmen Alpha-Beta, PVS, MTD(f), und MCTS interpretiert und analysiert.

Diese in Kapitel 5 durchgeführten Tests sollen so einen umfassenden Überblick in die jeweiligen Stärken und Schwächen dieser Algorithmen im Kontext des Spiels Amazons geben. Dadurch wurde nicht nur ein systematischer Vergleich durchgeführt, sondern stimmt auch weitgehend mit bestehender Literatur überein und bestätigen teilweise frühere Arbeiten in diesem Bereich. Jedoch werden auch überraschende Verhaltensweisen einiger Algorithmen unter bestimmten Testbedingungen offenbart.

6.1 Iterationsergebnisse mit 60 und 30 Sekunden Zeitlimit

Bei einem Zeitlimit von 60 Sekunden zeigt MTD(f) unter den Alpha-Beta-Varianten die geringste Anzahl an Iterationen pro Zug. Dies ist auf das erhöhte Pruning durch den Nullfenstersuchansatz von MTD(f) zurückzuführen, wie es auch in den Studien von Kloetzer et al. (2008) beobachtet wurde, die MTD(f) insbesondere für endspielartige Szenarien als besonders effektiv erkannten [2]. PVS weist eine höhere Iterationszahl auf als Alpha-Beta, was auf die tiefergehende Suche in der Hauptvariante zurückzuführen ist, da PVS versucht, den kritischen Zug durch eine tiefere Vorwärtssuche zu identifizieren. Diese Ergebnisse entsprechen den Arbeiten von Qiu et al. (2014) [7]. MCTS zeigt aufgrund des hohen Verzweigungsfaktors des Spiels im frühen Stadium des Spiels eine geringe Iterationszahl, die jedoch mit zunehmendem Spielverlauf und reduzierter Zuganzahl ansteigt.

Beim 30-Sekunden-Zeitlimit hat PVS nun mehr Iterationen pro Zug als Alpha-Beta sowohl in der Anfangsphase des Spiels als auch in der mittleren Spielphase, da PVS effizienter auf eine tiefere Suche in kritischen Varianten optimiert wurde und in kürzerer Zeit häufiger auf ähnliche Szenarien trifft, die schneller verarbeitet werden können. Wieder zeigt MTD(f) die geringste Iterationsanzahl, was erneut die starke Pruning-Wirkung widerspiegelt. Auffällig ist, dass PVS in der Mitte des Spiels die höchste Iterationszahl erreicht, was auf die Tiefe der Hauptvarianten zurückzuführen ist, die in dieser Phase besonders kritisch für den Spielverlauf sind.

6.2 Vergleich der Laufzeiten

Bei dem Vergleich der Laufzeiten wird deutlich, dass Alpha-Beta in den niedrigeren Tiefenstufen 1 und 2 die geringste Berechnungszeit benötigte, währenddessen bei der Tiefe 3 MTD(f) und bei der Tiefe 4 PVS kürzere Berechnungszeiten aufwiesen. Dadurch wird deutlich, dass MTD(f) durch den Nullfenstersuchansatz in tieferen Suchen effizienter wird und PVS durch die Hauptvariantenstrategie in fortgeschritteneren Suchtiefen effizienter wird. Diese Beobachtungen bestätigen die Erkenntnisse

der Studien von Qiu et al. (2014) und Bouzy et al. (2007), welche die Effizienz der tiefen Suchen von PVS und MTD(f) durch Optimierungen wie Hauptvarianten und Transpositionstabellen belegten [7, 1].

6.3 Effizienz in verschiedenen Spielphasen

In der frühen Spielphase, bei einem 60-Sekunden-Zeitlimit und der vorgegebenen Spielposition, fanden Alpha-Beta und PVS den stärksten Zug, wobei Alpha-Beta überraschenderweise die geringste Iterationsanzahl und PVS die höchste Iterationsanzahl aufwies. Diese Resultate lassen sich durch die Eigenschaften der Algorithmen erklären: PVS fokussiert sich auf die Hauptvariante und durchläuft im Vergleich mehr Iterationen, während Alpha-Beta im frühen Stadium effizientere Pruning-Techniken anwendet, was zur Reduktion der Iterationsanzahl beiträgt und dennoch starke Züge findet. Hier wird noch mal deutlich, wie effizient Alpha-Beta im frühen Stadium des Spiels arbeitet und Algorithmen wie MCTS und MTD(f) schlechter abschneiden, aufgrund des sehr großen Suchraumes.

In der mittleren Spielphase zeigten sich MTD(f) und PVS als effizienter, wobei MTD(f) die höchste Iterationszahl und PVS die geringste aufwies. Dies entspricht den Ergebnissen von J. Kloetzer et al. (2008) und Guo et al. (2019), die ebenfalls beobachteten, dass PVS und MTD(f) besonders für die detaillierte Analyse in taktisch anspruchsvollen Positionen gut geeignet sind [2, 11]. PVS verzichtet in dieser Phase verstärkt auf unnötige Knoten, während MTD(f) aufgrund seines Prunings gezielt eine hohe Iterationszahl nutzt, um den besten Zug zu identifizieren. Überraschenderweise schneidet Alpha-Beta hier am schlechtesten ab.

In der Endspielphase hat Alpha-Beta den besten Zug gefunden und benötigt hierbei die geringste Iterationsanzahl, während PVS, mit der höchsten Iterationsanzahl, am schwächsten abschneidet. MCTS konnte den zweit stärksten Zug finden, durch die erhöhte Iterationszahl (aufgrund des nun kleineren Suchraumes). Diese Ergebnisse zeigen, dass Alpha-Beta durch seinen robusten Pruning-Mechanismus gerade in Endspielphasen, in denen weniger Züge zur Verfügung stehen, besonders effizient arbeitet. Dadurch das MCTS in der mittleren und Endspielphase gut abschneidet und in der frühen am schlechtesten zeigt, dass Optimierungsbedarf gerade in der Anfangsphase besteht und der zu große Suchraum ein Problem darstellt.

6.4 Speicherbedarf der Transpositionstabelle

Die Analyse des Speicherbedarfs der jeweiligen Transpositionstabellen ergab, dass MTD(f) den größten Speicherbedarf entlang den verschiedenen Spielphasen aufwies, gefolgt von PVS und Alpha-Beta. Der hohe Speicherbedarf von MTD(f) zeigt die Notwendigkeit, zusätzliche Knoten zu speichern, um effektiver *Prunen* zu können, während Alpha-Beta durch gezieltes Speichern von Schlüsselzügen und selektivem *Pruning* eine vergleichsweise geringe Speicherlast aufweist. Dementsprechend speichern PVS und MTD(f) die meist bewerteten Positionen.

6.5 Siegquote der Algorithmen

Die Siegquote bei einem Zeitlimit von 60 Sekunden und 30 Sekunden zeigte, dass Alpha-Beta und PVS durchweg überlegen waren. Alpha-Beta erreichte die höchste Siegesquote, gefolgt von PVS und Monte Carlo Tree Search während MTD(f) das schwächste Ergebnis erzielte. Dies steht im Einklang mit der Literatur, die Alpha-Beta und PVS für deterministische Spiele wie Amazons favorisiert [14, 2]. MCTS konnte seine Siegquote bei einem Zeitlimit von 30 Sekunden im Vergleich zu 60 Sekunden leicht erhöhen, da durch die reduzierte Suchtiefe, das Entscheidungsfindungsverfahren erleichtert wurde. Mit weniger verfügbaren Simulationen pro Zug muss MCTS auf weniger mögliche Züge eingehen, was dazu führt, dass sich das Entscheidungsverfahren stärker auf die besten sofort erkennbaren Optionen konzentriert. Dadurch kann der Algorithmus möglicherweise schneller die oberflächlich besten Züge priorisieren, anstatt tiefere, komplexere Positionen zu explorieren, die mit zusätzlicher Zeit oft umfangreiche Berechnungen und eine potenziell größere Varianz erfordern.

6.6 Asymmetrischer Test MCTS vs Alpha-Beta

Der asymmetrische Test zeigt, das selbst bei einem Zeitlimit von 90 Sekunden für den MCTS und für Alpha-Beta ein Zeitlimit von nur 10 Sekunden pro Zug, dass MCTS dennoch lediglich nur 25% der Spiele gewinnen konnte. Dieses Ergebnis kann darauf zurückgeführt werden, dass MCTS trotz des längeren Zeitlimits (was eine erhöhte Anzahl von Simulationen pro Zug bedeutet) im Vergleich zu Alpha-Beta bei deterministischen Spielen ohne hybride Ansätze oder optimierte Bewertungsfunktionen weniger effizient arbeitet als Alpha-Beta und PVS [6]. Denn ein Experiment zeigte, dass die Erhöhung der Anzahl von Simulationen in einem MCTS-Programm nicht zwangsläufig zu einer Verbesserung der Spielstärke führt, insbesondere im Vergleich zu klassischen Alpha-Beta-Pruning-Verfahren [14]. Die Ergebnisse deuten darauf hin, dass eine Kombination von MCTS mit einer verbesserten Bewertungsfunktion und hybridem Ansatz effektiver sein kann als das bloße Erhöhen der Simulationen. Dies zeigt, dass die Bewertungsfunktion eine zentrale Rolle für den MCTS im Kontext von Amazons spielt.

6.7 Limitationen der Arbeit

Eine wesentliche Limitierung dieser Arbeit ist die Beschränkung auf das Spiel Amazons und ein enges Zeitlimit. Die Ergebnisse sind daher nur bedingt auf andere Spiele mit unterschiedlichen Zugstrukturen und Komplexitäten übertragbar. Zudem könnten die Ergebnisse bei dynamischen Zeitbegrenzungen, Verwendung einer anderen Programmiersprache oder spezifische Anpassungen an die Bewertungsfunktion abweichen. Zusätzlich gibt es noch eine Begrenzung der MCTS-Optimierungen. Obwohl verschiedene Simulationstiefen und Zugpriorisierungen getestet wurden und Optimierungen an dem MCTS vorgenommen wurden, wäre es interessant gewesen, eine Hybridstrategie zu untersuchen, die MCTS mit deterministischen Algorithmen wie Alpha-Beta kombiniert, wie von Jianning et al. (2015) [13] vorgeschlagen und ebenfalls in der Arbeit von Richard J. Lorentz diskutiert wird [6].

6.8 Beitrag der Arbeit

Die Arbeit bestätigt die bestehenden Forschungsergebnisse zu den Suchalgorithmen im Spiel Amazons. Durch den systematischen Vergleich der Algorithmen Alpha-Beta, PVS, MTD(f) und MCTS wurden die Effizienzunterschiede in verschiedenen Spielsituationen verdeutlicht. Die durchgeführten Untersuchungen bieten eine Grundlage für zukünftige Optimierungen insbesondere für den Monte Carlo Tree Search. Denn durch diese Arbeit wurde noch mals gezeigt, dass eine hybride Strategie für den MCTS entscheidend ist und weitere Optimierungspotenziale bestehen, wie auch das Wiederverwenden eines Teilbaumes. Diese Erkenntnisse können zu einer Weiterentwicklung der jeweiligen Suchalgorithmen im Kontext der Spieltheorie beitragen.

7 Fazit

Diese Arbeit befasste sich mit dem systematischen Vergleich und der genaueren Analyse, im Kontext des Brettspiels Amazons, von den vier Algorithmen: Alpha-Beta, Principal Variation Search (PVS), MTD(f) und Monte Carlo Tree Search (MCTS). Die Forschungsfrage befasste sich mit der Frage, welche Stärken und Schwächen diese Algorithmen in den verschiedenen Spielphasen und unter verschiedenen Bedingungen mit sich bringen, sodass bestimmt werden kann, welcher dieser Ansätze in der komplexen Spielumgebung Amazons die beste Leistung erbringt.

Die Ergebnisse zeigen, dass Alpha-Beta und PVS besonders in frühen und mittleren Spielphasen starke Leistungen erzielen, da diese Algorithmen effizient im Pruning (Beschneiden) des Suchbaumes sind und durch gezielte Heuristiken die relevantesten Züge priorisieren. MTD(f) zeigt seine Stärke in Endspiel-Szenarien, wo er durch effizientes Pruning die Anzahl an notwendigen Iterationen stark reduzieren kann. MCTS hingegen schnitt im direkten Vergleich mit den Minimax-basierten Algorithmen erwartungsgemäß schlechter ab, konnte jedoch bei kürzeren Zeitlimits seine Siegquote leicht verbessern.

Diese Arbeit bestätigt, dass klassische Minimax-basierte Algorithmen, insbesondere Alpha-Beta und PVS, durch optimierte Suchtiefe und effiziente Heuristiken im Brettspiel Amazons insgesamt effektiver sind als stochastische Suchstrategien wie Monte Carlo Tree Search. Die Arbeit zeigt ebenfalls, dass MCTS zwar für komplexe und hochverzweigte Spielbäume wie in Amazons geeignet ist, in diesem Kontext aber von gezielten Optimierungen der Playouts, Wiederverwendung von Teilbäumen und hybriden Ansätzen sehr stark profitieren könnte.

Zukünftige Forschungsfragen, die sich aus dieser Arbeit ergeben, umfassen die Untersuchung hybrider Algorithmen, die MCTS und Minimax-basierte Methoden kombinieren, um die jeweiligen Stärken beider Ansätze zu kombinieren. Zudem könnten weiterführende Untersuchungen der Algorithmen unter unterschiedlichen Spielbrettkonfigurationen und adaptiven Zeitlimits interessante Einblicke in deren Anpassungsfähigkeit und Effizienz bieten.

Abschließend wurde gezeigt, dass die Wahl der geeigneten Suchstrategie für das Brettspiel Amazons von der Spielphase und den verwendeten Optimierungen und damit zur Verfügung stehende Ressourcen abhängt. Minimax-basierte Algorithmen wie Alpha-Beta und PVS bieten bei einer optimalen Konfiguration eine überlegenere Leistung während MCTS als ein potenzieller Kandidat für innovative und hybride Ansätze bestehen bleibt.

Hilfsmittel

1. *Für die gesamte Arbeit wurde für die Grammatik- und Rechtschreibprüfung das generische KI-Tool ChatGPT (Version 4o) von OpenAI verwendet.*

2. *Zusätzlich wurde für das Erstellen unterschiedlichen Spielpositionen zwar das Ursprungsbild von Wikipedia verwendet [12], jedoch mithilfe von Paint 3D verändert (Abbildungen Nr. 2.2, 2.3, 2.4 und 2.5).*

3. *Sonstige Grafiken, wie die vier Phasen des MCTS (3.2) oder Alpha-Beta-Pruning Grafik (3.1) wurden mit draw.io erstellt.*

Literaturverzeichnis

[1] Julien Kloetzer, Hiroyuki Iida, Bruno Bouzy. The monte carlo approach in amazons. In *International Conference on Computers and Games*, pages 185–192. Springer, 2007.

[2] Julien Kloetzer, Hiroyuki Iida, Bruno Bouzy. A comparative study of solvers in amazons endgames. In *IEEE Symposium on Computational Intelligence in Games*, pages 140–148. IEEE, 2008.

[3] Aske Plaat, Jonathan Schaeffer, Wim Pijls, Arie de Bruin. A new paradigm for minimax search. pages 5–20. Self-published, 1994.

[4] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. pages 97–109. ScienceDirect, 1985.

[5] Jens Lieberum. An evaluation function for the game of amazons. pages 1–7. Springer, 2005.

[6] Richard J. Lorentz. Amazons discover Monte–Carlo. In *Computers and Games*, pages 25–36. Springer, 2008.

[7] Hongkun Qiu, Peng Zhang, Yajie Wang, Jiehong Wu, and Fei Li. Analysis of search algorithm in computer game of amazons. In *2014 IEEE International Conference on Computer Games*, pages 3947–3950. IEEE, 2014.

[8] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. In *Transactions on Pattern Analysis and Machine Intelligence*, pages 701–710. IEEE, 1994.

[9] Steven James, George Konidaris, Benjamin Rosman. An analysis of monte carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3576–3583. 2017.

[10] Bairui Tong, Hongkun Qiu, Tiantian Guo, Yajie Wang. Research and application of parallel computing of pvs algorithm based on amazon human-machine game. pages 1–6. IEEE, 2019.

[11] Tiantian Guo, Hongkun Qiu, Bairui Tong, Yajie Wang. Optimization and comparison of multiple game algorithms in amazon chess. pages 1–6. IEEE, 2019.

[12] Wikipedia. Game of the amazons. https://en.wikipedia.org/wiki/Game_of_the_Amazons. Zuletzt besucht: 22.08.2024.

[13] Quan Jianning, Qiu Hongkun, Wang Yajie, Li Yan, Wang Xiaoyan. Study the performance of search algorithms in amazons. In *IEEE 27th Chinese Control and Decision Conference (CCDC)*, pages 5812–5813. IEEE, 2015.

[14] Hikari Kato, Szilárd Zsolt Fazekas, Mayumi Takaya, Akihiro Yamamura. Comparative study of monte-carlo tree search and alpha-beta pruning in amazons. In *3rd International Conference on Information and Communication Technology-EurAsia (ICT-EURASIA)*, pages 139–148. Springer, 2015.

[15] Gün Yanik. Heuristische vs. stochastische suche - alphabeta pruning und monte carlo tree search im spiel der amazonen. page 18. Technische Universität Berlin, 2021.

[16] Walter Zamkauskas. Amazons. In *The Game of Amazons*, pages 1–13. Self-published, 1988.