Analyse eines Minimax/MCTS Hybriden anhand des Spiels „Catch the Lion“

Bachelorarbeit

Bruno Schaffer
414062

30. April 2024

Betreuer: 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

Monte-Carlo-Tree-Search ist in den letzten Jahren ein weit verbreiteter Spielalgorithmus geworden, insbesondere aufgrund seines Erfolgs im Spiel „Go“. Dennoch hat der Algorithmus Schwierigkeiten, unauffällige Zugfolgen zu erkennen, die zu einer Niederlage führen können. Um diese Schwäche auszugleichen, wurden von Hendrik Baier und Mark H. M. Winands die „MCTS-Minimax Hybrids“ vorgeschlagen. In dieser Arbeit wird die von Baier und Winands gezeigte Spielstärke der Hybridalgorithmen überprüft, indem die Algorithmen im Spiel „Catch the Lion“ gegen unterschiedliche Algorithmen getestet werden. „Catch the Lion“ eignet sich besonders gut, da es mit einer hohen Anzahl von *sudden deaths* das ideale Spiel für einen MCTS-MiniMax-Hybridalgorithmus ist.

Wie erwartet wurde, konnten in den Spielexperimenten eine erhöhte Spielstärke der MCTS-MiniMax-Hybridalgorithmen im Vergleich mit dem Basis-Monte-Carlo-Tree-Search-Algorithmus festgestellt werden. Dennoch ist auch klar geworden, dass trotz der höheren Spielstärke die Alpha-Beta-Suche der bessere Algorithmus im Spiel „Catch the Lion“ ist. MCTS-MiniMax-Hybride sind trotzdem eine gute Wahl in Spielen mit erhöhter Anzahl an *sudden deaths*. Ist diese hohe Anzahl jedoch in Kombination mit einem relativ geringen *branching factor*, dann ist es sinnvoller, die Alpha-Beta-Suche direkt zu verwenden, anstatt sie nur als Verbesserung in Monte-Carlo-Tree-Search zu implementieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In den letzten Jahren hat der Monte-Carlo-Tree-Search-Algorithmus (MCTS) aufgrund von Durchbrüchen im Bereich der Spielalgorithmen an Bekanntheit gewonnen, besonders im Spiel „Go“. Dieses galt lange Zeit als zu komplex für Computerprogramme, da es einen deutlich größeren *branching factor* als Spiele wie Schach aufweist, in denen Algorithmen wie Alpha-Beta bereits gut funktionierten. [8] Im Jahr 2016 gelang es jedoch Google DeepMinds AlphaZero, den Top-Go-Spieler Lee Sedol mit 4:1 zu besiegen. [2] Um diesen Sieg zu erreichen, kombinierte AlphaZero Monte-Carlo-Tree-Search mit Deep Learning. [3] Der Sieg von AlphaZero war ein großer Durchbruch. AlphaZero wurde sogar der höchste Go-Großmeister-Rang von „9 dan“ von der Korea Baduk Association verliehen. [2]
Dennoch hat die Monte-Carlo-Tree-Search einen Nachteil, der möglicherweise der Grund dafür ist, dass AlphaZero gegen Lee Sedol in der vierten Partie verloren hat. Der Nachteil besteht darin, dass in bestimmten Positionen Züge fälschlicherweise stark wirken und eine unauffällige Zugfolge verbergen können, die zur Niederlage führt. Solche *sudden death* Positionen müssen sorgfältig analysiert werden. Aufgrund der Auswahlstrategie von Monte-Carlo-Tree-Search ist es allerdings möglich, dass der Algorithmus diese Fallen übersieht.[15] Insbesondere in Spielen wie Schach oder „Catch the Lion“ potenziert sich dieses Problem, da eine Niederlage oft nur wenige falsche Züge entfernt ist. Dies steht im Gegensatz zu „Go“, wo sich das Spiel erst nach vielen Zügen entscheidet. Daher ist die Monte-Carlo-Tree-Search oft noch schlechter als MiniMax in Spielen, die eine große Anzahl an terminalen Zuständen und *sudden deaths* aufweisen [4].

Literatur

Dieser Vorteil von MiniMax gegenüber der Monte-Carlo-Tree-Search wirft die Frage auf, ob es möglich ist, den Nachteil der Monte-Carlo-Tree-Search durch Integration von MiniMax zu verbessern. Dieser Ansatz wurde von Hendrik Baier und Mark H. M. Winands in „MCTS-Minimax Hybrids“ analysiert [4].
Um diese Verbesserung überhaupt möglich zu machen, ist es notwendig, den Umgang mit terminalen Spielzuständen von Monte-Carlo-Tree-Search durch die *MCTS-Solver Extension* von Mark H. M. Winands, Yngvi Björnsson und Jahn-Takeshi Saito zu verbessern. Die *MCTS-Solver Extension* wurde in „Monte-Carlo Tree Search Solver“ vorgestellt.[23]

Ziel dieser Arbeit

Das Ziel dieser Arbeit ist es daher, anhand des Spiels „Catch the Lion“ zu überprüfen, ob es möglich ist, die Monte-Carlo-Tree-Search durch die MiniMax-Suche zu verbessern. Laut den Ergebnissen aus „MCTS-Minimax Hybrids“ [4] ist eine Verbesserung des Monte-Carlo-Tree-Search-Algorithmus definitiv zu erwarten. Ob ein MCTS-MiniMax-Hybrid auch besser als die normale MiniMax-Suche ist, wird sich zeigen. Da „Catch the Lion“ jedoch eine hohe Anzahl an *sudden deaths* aufweist [4], ist es durchaus möglich, dass MiniMax, insbesondere mit *Alpha-Beta-Pruning*, weiterhin der bessere Spielalgorithmus im Spiel „Catch the Lion“ ist.

Aufbau der Arbeit

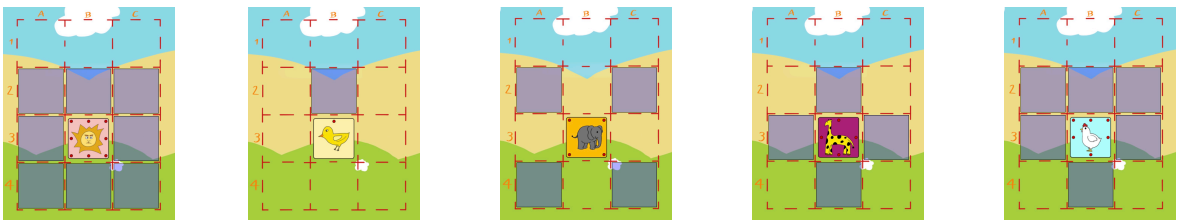
Zunächst werde ich in Kapitel 2 „Methoden“ das Spiel „Catch the Lion“ erklären sowie dessen Implementierung mittels Bitboards vorgestellt. Außerdem werden die relevanten Algorithmen erläutert. Einerseits wird der MiniMax-Algorithmus zusammen mit einigen seiner Verbesserungen wie *Alpha-Beta-Pruning*, Transpositionstabellen und MTD(f) behandelt. Andererseits wird der Monte-Carlo-Tree-Search-Algorithmus (MCTS) und die MCTS-Solver-Erweiterung erklärt. Zum Schluss werden diese beiden Algorithmen zu mehreren MCTS-MiniMax-Hybriden vereint. Dabei entstehen drei verschiedene Hybriden: MCTS mit MiniMax in der *Simulation*-Phase, MCTS mit MiniMax in der *Backpropagation*-Phase und MCTS mit MiniMax in der *Selection*-Phase.
In Kapitel 3 „Ergebnisse“ werden zunächst die verschiedenen Varianten des MiniMax-Algorithmus auf ihre Spielstärke getestet, gefolgt von ausführlichen Tests des MCTS-MiniMax-Hybriden. Das Ziel ist es, festzustellen, ob die Hybrid-Algorithmen eine Verbesserung gegenüber dem MCTS-Solver-Algorithmus darstellen. Dabei werden die Hybriden mit dem MCTS-Solver und dem Alpha-Beta-Algorithmus verglichen.
In Kapitel 4 „Diskussion“ werden die Ergebnisse aus Kapitel 3 analysiert. Dabei wird auf mögliche Gründe für die Ergebnisse hingewiesen und diskutiert, ob die Ergebnisse dieser Arbeit die Ergebnisse aus dem Paper ‚MCTS-Minimax Hybrids“ [4] bestätigen können. Insbesondere wird betrachtet, ob die Ergebnisse der Gegenüberstellung mit Alpha-Beta auch die Schlüsse aus dem Paper bestätigen können, da dieser Vergleich im Paper nicht durchgeführt wurde.
Im letzten Kapitel „Fazit“ wird die gesamte Arbeit reflektiert und eingeordnet. Es wird ebenfalls ein Blick auf die Themen geworfen, die nicht im Rahmen der Arbeit behandelt wurden, aber zukünftig erforscht werden sollten, wird ebenfalls geworfen. Abschließend wird festgestellt, welcher Algorithmus sich im Rahmen dieser Arbeit als am spielstärksten im Spiel „Catch the Lion“ erwiesen hat.

2 Methoden

2.1 „Catch The Lion“

2.1.1 Erläuterung des Spiels „Catch The Lion“

„Catch The Lion“ oder auch „Dōbutsu shōgi“ ist eine Variante des japanischen Schachspiels „Shogi“. „Catch the Lion“ wurde von der professionellen „Shogi“ Spielerin Kitao Madoka entwickelt, um Kinder an das Spiel heranzuführen. [9] Daher ist das Spiel simpler gehalten als normales „Shogi“. Während man „Shogi“ auf einem 9x9 Brett spielt, wird „Catch the Lion“ nur auf einem 3x4 Brett gespielt.
Trotzdem behält „Catch the Lion“ die meisten Regeln von „Shogi“ bei. Das Spiel wird mit fünf Figuren gespielt: Löwe, Giraffe, Elefant, Hühnchen und Henne. Das Ziel des Spiels ist es, den gegnerischen Löwen zu schlagen. Dabei bewegen die Spieler ihre Figuren abwechselnd. Jede Figur hat ein eigenes Bewegungsmuster, kann sich aber nur ein Feld weit bewegen (siehe Abbildung 2.1). Das Spiel kann außerdem auch gewonnen werden, wenn man den eigene Löwen auf die letzte gegnerischen Reihe zieht.
Die Drop-Regel aus „Shogi“ gibt es auch in „Catch the Lion“, diese besagt, dass, wenn man eine gegnerischen Figur schlägt, sie in die eigene Reserve kommt. Figuren in Reserve können auf jedem freien Spielfeld platziert werden. Außerdem gilt, dass, wenn ein Hühnchen die letzte Reihe erreicht, es zur Henne aufgewertet wird. Wenn die Henne geschlagen wird, kann sie jedoch nur als Hühnchen wieder platziert werden.
„Catch the Lion“ ist ein gelöstes Spiel. Das bedeutet, dass der theoretisch beste Zug für jede der 1.567.925.964 möglichen Postionen bekannt ist. „Catch the Lion“ beginnt in einer Zugzwangsituation, in welcher der Spieler, welcher als erstes ziehen muss, im Nachteil ist. Somit hat der Spieler, der als zweites zieht, eine Strategie, mit der er theoretisch nach maximal 78 Zügen gewinnt.[9] Eine weitere wichtige Eigenschaft von „Catch the Lion“ ist, dass das Spiel jederzeit durch das Schlagen des Löwen beendet werden kann. Die Anzahl der *sudden deaths* in „Catch the Lion“ ist somit größer als bei Spielen wie „Breakrough“, „Vier Gewinnt“ oder „Othello“ [4]



(a) Löwe (b) Hühnchen (c) Elefant (d) Giraffe (e) Henne

Abbildung 2.1: Möglichen Züge der Figuren (Quelle: eigene Abbildung)

2.1.2 Implementierung via *Bitboards*

„Catch the Lion“ und Schach sind sich vom grundlegenden Spielprinzip sehr ähnlich. Daher ist es möglich, sich bei der Implemetierung von „Catch the Lion“ an populären Strategien für die Implemetierung von Schach zu orientieren. Namentlich gibt es den *Mailbox*-Ansatz und den *Bitboard*-Ansatz.

Bei der Implementierung via *Mailbox* wird jedes Spielfeld als ein seperat addressierbares Speicherelement repräsentiert; oft innerhalb eines ein- oder zweidimensionalen Arrays, auf dem die Figuren als Objekte bewegt werden. Der Vorteil des *Mailbox*-Ansatz ist, dass er relativ leicht zu implementieren ist. Ein Nachteil ist jedoch, dass besonders die Berechnung der möglichen Züge rechenintensiv sein kann. [13] Für diese Arbeit wurde deshalb der *Bitboard*-Ansatz gewählt.
Die Idee von *Bitboards* besteht darin, das Spielbrett und die Figuren mittels Bit-Zahlen zu repräsentieren. [6] Im Fall von „Catch the Lion“ benötigt man 12 Bits pro *Bitboard*, ein Bit für jedes der 12 möglichen Felder. Um alle Figuren auf dem Brett repräsentieren zu können, sind 7 *Bitboards* nötig: 5 für die Figuren Löwe, Giraffe, Elephant, Hühnchen und Henne, sowie 2 *Bitboards*, um jeweils Schwarz und Weiß darzustellen. Das sind theoretisch nur 84 Bits.
Um zum Beispiel die Postion des weißen Löwen abzufragen, reicht es, das weiße *Bitboard* mit dem der Löwen zu verunden und man erhält ein *Bitboard*, welches die Position des weißen Löwen repräsentiert (siehe Abbildung 2.2).

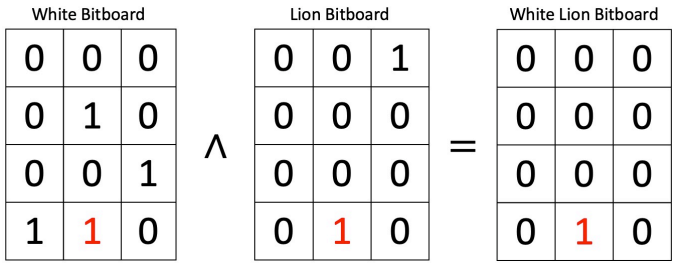


Abbildung 2.2: Weißer Löwe *Bitboard* (Quelle: eigene Abbildung)

Die Berechnung der möglichen Züge ist im Fall von „Catch the Lion“ mittels *Bitboards* ebenfalls recht einfach. Da jede Figur sich immer nur ein Feld weit bewegen kann, ist es möglich, für jede Figur die Bewegungsmuster für alle 12 Felder zu speichern. So reicht es, das Bewegungsmuster für das Feld auf dem die Figur steht mit dem negierten *Bitboard* des Teams der Figur zu verunden und man erhält die möglichen Züge für die Figur (siehe Abbildung 2.3).

Im Fall von „Catch the Lion“ gibt es noch die zu repräsentierenden Figuren in Reserve. Innerhalb dieser Arbeit werden zwei Listen genutzt, um die Reserve für jeweils Schwarz und Weiß zu speichern. Da Figuren aus der Reserve auf jedem freien Feld platziert werden können, ist die Berechnung der möglichen Züge für Figuren in Reserve durchaus einfach. Die freien Felder berechnen sich, indem man das schwarze und weiße *Bitboard* negiert und diese dann noch verundet (siehe Abbildung 2.4).

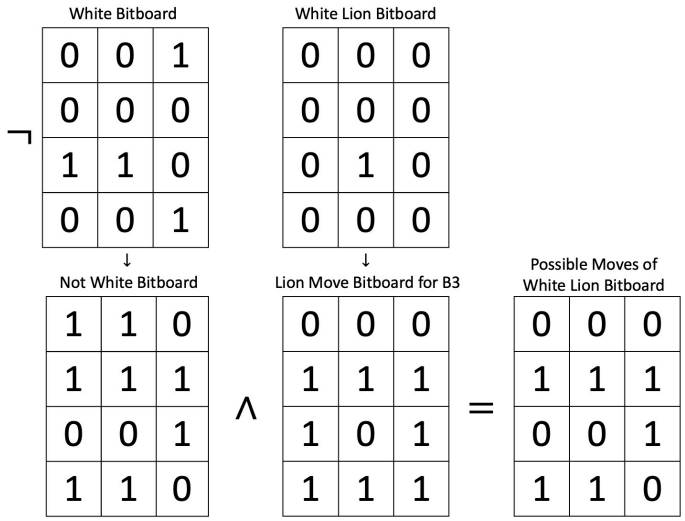


Abbildung 2.3: Mögliche Züge (Quelle: eigene Abbildung)

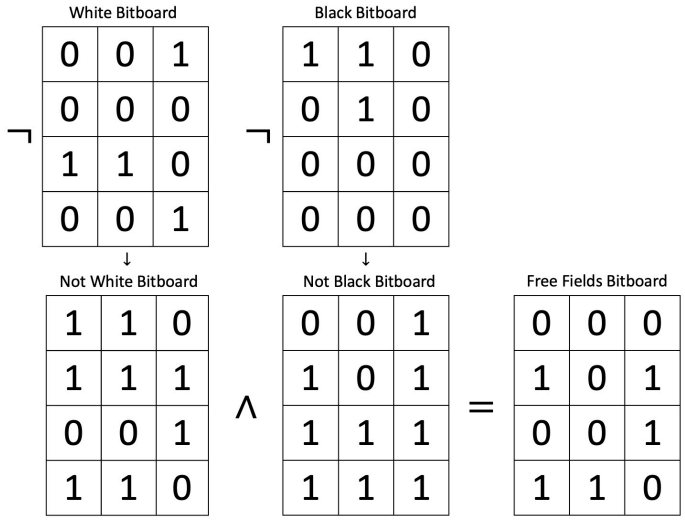


Abbildung 2.4: Freie Felder (Quelle: eigene Abbildung)

2.2 MiniMax-Algorithmen

2.2.1 MiniMax-Suche

Ein populärer algorithmischer Ansatz für Zwei-Personen-Nullsummenspiele wie „Catch The Lion“ ist die MiniMax-Suche. Die Idee der MiniMax-Suche besteht darin, den gesamten Lösungsbaum eines Spiels bis zu einer festen Tiefe zu erkunden.
Dabei wird der Spielstand mittels eines Evaluationswerts repräsentiert. Wenn dieser Wert positiv ist, zeigt er beispielsweise einen Vorteil für den weißen Spieler. Ist der Wert jedoch negativ, repräsentiert er einen Vorteil für den anderen Spieler, beispielsweise für den schwarzen Spieler. Der Betrag des Werts gibt zusätzlich den Grad des Vorteils für den jeweiligen Spieler an. 0,5 bedeutet beispielsweise einen kleinen Vorteil, während 15 einen großen Vorteil für den weißen Spieler darstellt. Das Ziel des einen Spielers ist es daher, den Wert zu maximieren (Max-Spieler), während das Ziel des anderen Spielers darin besteht, den Wert zu minimieren (Min-Spieler).
Der Algorithmus funktioniert, indem er sich selbst rekursiv aufruft bis die festgelegte Tiefe erreicht ist und der berechnete Evaluationswert zurück propagiert werden kann. Bei der Rückpropagation wählen der Max- und Min-Spieler abwechselnd den Zug mit dem jeweils minimalen oder maximalen Wert aus, sodass am Ende der beste Zug mit dem optimalen MiniMax-Wert gewählt wird. [22, vgl. Kapitel 7.1]

2.2.2 *Alpha-Beta-Pruning*

Während der Erkundung des MiniMax-Suchbaums ist es möglich, dass Teile des Suchbaums nicht erkundet werden müssen, da in einem anderen Teil des Baums bereits ein besserer Zug gefunden wurde. Zum Beispiel hat der Max-Spieler an einer Stelle zwei mögliche Züge A und B. Für Zug A wurden bereits alle Möglichkeiten betrachtet und der Zug wurde mit dem Wert 3 bewertet. Während der Betrachtung von Zug B kommt der Min-Spieler auf den Wert -5 für einen Teil des Suchbaums. Nun ist es zwar möglich, den Baum weiter zu erkunden, jedoch hat der Min-Spieler bereits die Option, einen Zug mit dem Wert -5 zu wählen. Ihn interessieren also nur noch Züge mit einer Bewertung < -5. Da jedoch der darüber liegende Max-Spieler bereits einen Zug mit der Bewertung 3 wählen kann, ist es nun egal, ob der Min-Spieler noch einen Zug mit einer Bewertung < -5 findet. Da der Max-Spieler immer den Zug mit der Bewertung 3 wählen wird, kann also die Suche für Zug B nach dem Finden des Zuges mit der Bewertung -5 beendet werden, wie in Abbildung 2.5 zu sehen ist. [22, vgl. Kapitel 7.2]

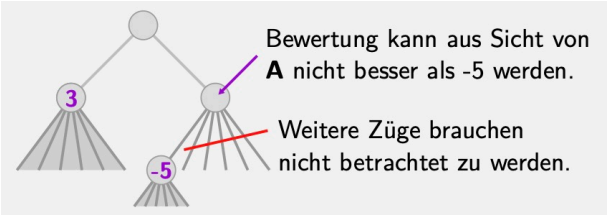


Abbildung 2.5: *Alpha-Beta-Cutoff* (Quelle:Algorithmen und Datenstrukturen [22, Kapitel 7.2])

Um dieses Prinzip algorithmisch zu implementieren, reicht das Hinzufügen der beiden namensgebenden Parameter Alpha und Beta aus. Hier repräsentiert Alpha den besten Zug, den der Max-Spieler zur Verfügung hat, und Beta den besten Zug, den der Min-Spieler zur Verfügung hat. Die Werte werden während der Suche stetig aktualisiert. Wenn nun diese beiden Schranken kollidieren, also Alpha >= Beta, heißt es, die Suche für den Teilbaum kann beendet und der aktuelle Wert zurückgegeben werden. [22, vgl. Kapitel 7.2]

2.2.3 NegaMax Variante

Bei der Programmierung des MiniMax-Algorithmus fällt eine Sache deutlich auf: Die Implementierung des Min- und Max-Spielers ist beinahe identisch. Es sollte also möglich sein, die Implementierung der beiden Spieler zusammenzufassen. Mit dem NegaMax-Ansatz ist das möglich; die beiden Spieler werden zu einem maximierenden Spieler zusammengeführt. Um jedoch weiterhin den Wechsel zwischen zwei Spielern zu simulieren, wird das Vorzeichen des zurückgegebenen Bewertungswerts mit steigender Suchtiefe alterniert. Somit ist der Wechsel zwischen Min- und Max-Spieler weiterhin gewährleistet, da die Maximierung der negierten Werte nichts anderes als die Minimierung der normalen Werte ist. Wenn *Alpha-Beta-Pruning* implementiert ist, ist es außerdem notwendig, beim rekursiven Aufruf Alpha und Beta zu vertauschen und zu negieren. [22, vgl. Kapitel 7.2.1]

2.2.4 Alpha-Beta mit Transpositionstabellen

Im Spiel „Catch The Lion“ ist es möglich, durch unterschiedliche Zugreihenfolgen die gleiche Spielstellung zu erreichen. Ein einfaches Beispiel wäre, wenn beide Spieler ihren Löwen bewegen und ihn im nächsten Zug wieder zurück bewegen. So kann es also vorkommen, dass beim Erkunden des MiniMax-Suchbaums die gleiche Spielstellung mehrfach berechnet wird. Es liegt also nahe, bereits erkundete Spielstellungen in einer Datenbank zu speichern, anstatt sie jedes Mal neu zu erkunden. Diese Optimierung nennt man Transpositionstabellen. [20] Bei der Integration von Transpositionstabellen in den Alpha-Beta-Algorithmus gibt es mehrere Dinge, die beachtet werden müssen: das Speichern und Auslesen der Werte sowie die Implementierung der Datenbank selbst.

Das Speichern der Werte innerhalb des Basis-MiniMax-Algorithmus ist unkompliziert, da alle Ergebnisse als genaue Übereinstimmung gespeichert werden können. Durch die Einführung der *Cutoffs* ist das im Alpha-Beta-Algorithmus nicht der Fall. Kommt es zum *Alpha-Cutoff*, reicht die Information nur, um den MiniMax-Wert als obere Grenze zu speichern und beim *Beta-Cutoff* als untere Grenze. [18, siehe AlphaBetaWithMemory Pseudo Code]
Dementsprechend ist es auch nötig, beim Auslesen der Werte darauf zu achten, ob es sich um eine genauen gespeicherten Wert oder nur eine untere oder obere Grenze handelt. Im Fall einer genauen Übereinstimmung kann der MiniMax-Wert direkt zurückgegeben werden. Die untere und obere Grenze reichen nur aus, um jeweils den Wert von Alpha und Beta anzupassen. Außerdem ist es notwendig, nur Transpositionstabelleneinträge mit einer Tiefe zu verwenden, die zur Tiefe des aktuellen MiniMax-Calls passt. [18, siehe AlphaBetaWithMemory Pseudo Code]
Um eine Spielstellung und den dazugehörigen MiniMax-Wert in einer Datenbank zu speichern, braucht man einen Index für die Datenbank. Zur Generierung dieses Index benötigt man eine Form von Hashing; innerhalb dieser Arbeit wurde das *Zobrist-Hashing* verwendet. Da Datenbanken normalerweise nicht groß genug sind, um alle Spielstellungen zu speichern, kann es zu Kollisionen kommen. Die berechnete Datenbankadresse von zwei Spielständen ist hier identisch und es können nicht beide gespeichert werden. Hier wäre es notwendig, sich ein Verfahren zur Bewältigung von Kollisionen zu überlegen. [20]

Mithilfe von Transpositionstabellen kann Alpha-Beta deutlich verbessert werden, sodass für die gleiche Suchtiefe weniger Zeit benötigt wird. Außerdem ist es ein notwendiges Verfahren für weitere Verbesserungen wie MTD(f). [18]

2.2.5 *Zobrist-Hashing*

Das Ziel des *Hashing* ist es, jede Spielstellung in eine fast eindeutige Index-Zahl von fester Länge zu transformieren. Hierbei ist es wichtig, dass zwei ähnliche Stellungen einen komplett unterschiedlichen Index erhalten.[25]
Beim *Zobrist-Hashing* wird zu Beginn des Programms ein Array an *Pseudorandom*-Zahlen generiert. Für jede einzelne Figur wird eine Zahl pro Feld auf dem Spielbrett generiert, also zum Beispiel für den weißen Löwen zwölf Pseudorandom-Zahlen, eine für jedes der zwölf Felder. Zudem wird noch eine Zahl generiert, welche repräsentiert, ob es der Zug des schwarzen Spielers oder des weißen Spielers ist. Zur Repräsentation der Figuren, welche sich in der Reserve befinden, werden jeweils sechs weitere Zahlen benötigt, je nachdem, ob zum Beispiel ein Elefant in der Reserve ist. [25]
Um nun den Zobrist-Hash zu berechnen, reicht es aus, alle zum Spielstand passenden Zahlen zu verXORen. Zum Beispiel:

```
(Weißer Löwe auf B3) xor (Schwarzes Huhn auf A2) ... xor (Schwarz am Zug) ...
xor (Huhn in weißer Reserve) = Zobrist-Hash
```

Während der Ausführung eines Spielalgorithmus, zum Beispiel die Alpha-Beta-Suche, kommt es oft dazu, dass ein Zug gemacht, etwas berechnet und dann dieser Zug wieder rückgängig gemacht wird. Hier wäre es möglich, für jede Spielposition den Hash komplett neu zu berechen, das ist jedoch mit Rechenaufwand verbunden. Da jedoch die vom *Zobrist-Hashing* genutzte XOR-Operation auch die Umkehrfunktion für XOR ist, ist es leicht, eine Figur aus der Repräsentation zu löschen, indem man einfach nochmal dieselbe XOR-Operation anwendet. Somit ist es möglich, schnell aus dem bestehenden Hash den Hash für die neue Position zu berechnen. Hierfür reicht es, die alte Figurposition mit XOR aus dem Hash zu entfernen und die neue Figurposition mit XOR hinzuzufügen. Hier zu sehen anhand dem Beispiel eines Zuges des Huhns von A2 nach A1: [25]

```
(Bestehenden Hash) xor (Schwarzes Huhn auf A2) xor (Schwarzes Huhn auf A1)
= neuer Zobrist-Hash
```

Um nun den Zug wieder rückgängig zu machen, reicht es, die gleichen Operationen wie zum Durchführen des Zuges auf den Hash wieder anzuwenden.
Der Zobrist-Hash ermöglicht es, für jede Spielposition eine nahezu eindeutige Zahl zu generieren. Eine wichtige Eigenschaft dabei ist, dass der Hash-Wert für zwei ähnliche Positionen komplett unterschiedlich ist. *Zobrist-Hashing* eignet sich daher sehr gut als Index für schnelle und platzsparende Hash-Tabellen. [25]

2.2.6 Zugreihenfolge

Eine der größten Zeiteinsparungen der MiniMax-Suche sind die durch Alpha-Beta erzeugten *cutoffs*. Die Anzahl der *cutoffs* steigt, je früher der beste Zug evaluiert wird, da mit dem idealen Evaluationswert die meisten *cutoffs* erzeugt werden können. Das Ziel ist es, dem minimalen Suchbaum so nahe wie möglich zu kommen, indem man versucht, den besten Zug als erstes zu evaluieren. Zu diesem Zweck ist es also sinnvoll, die Züge zu sortieren. [14]
Als erstes sollte man Züge evaluieren, die zum Sieg führen. Dies geschieht beispielsweise durch das Schlagen des gegnerischen Löwen in „Catch The Lion“, da, wenn dieser Zug existiert, fast alle folgenden Züge nicht mehr evaluiert werden müssen. Nutzt man iteratives Deepening, dann ist es sinnvoll, die *principal variation* aus der letzten Iteration zu evaluieren, weil es die beste Einschätzung des optimalen Zuges ist, die man hat. Danach folgen Züge, die das Ergebnis der Evaluationsfunktion stark beeinflussen, also zum Beispiel Züge, bei denen im Spiel „Catch The Lion“ eine gegnerische Figur geschlagen wird. Zum Schluss folgen die restlichen Züge also zum Beispiel das normale Ziehen einer Figur.
Damit Alpha-Beta effizient läuft, ist eine gute Zugreihenfolge notwendig. [14]

2.2.7 *Iterative Deepening*

Die MiniMax-Suche ist in der normalen Ausführung nicht zeitbegrenzt, sondern durch begrenzte Suchtiefen limitiert. MiniMax liefert erst ein Ergebnis, wenn der komplette Suchbaum bis zur festgelegten Tiefe erkundet wurde. Der *branching factor* bleibt jedoch während des Verlaufs des kompletten Spiels nicht konstant. In Tabelle 2.1 sieht man, dass Startpositionen und Endspielpositionen große Unterschiede in der benötigten Rechenzeit aufweisen können. In Wettbewerben gibt es jedoch oft ein Zeitlimit, daher ist es keine valide Strategie, eine Suchtiefe für das gesamte Spiel festzulegen und darauf zu hoffen, dass das Zeitlimit nicht überschritten wird. Es ist eine Strategie erforderlich, die immer ein Ergebnis mit möglichst großer Tiefe innerhalb des Zeitlimits liefert.

Tabelle 2.1: Alpha-Beta Suchtiefe 5 (Quelle: eigene Tabelle)

Position	Rechenzeit (ms)
Startpostion	43.5
Mate in 3	15.6
Mate in 3	214.1

Iterative Deepening ist eine einfache Zeitmanagement-Strategie, die dieses Problem angeht. Das Konzept von *Iterative Deepening* ist durchaus unkompliziert. Der Grundgedanke besteht darin, die Suchtiefe iterativ zu erhöhen, solange noch Zeit zur Verfügung steht. Wenn das Zeitlimit erreicht wird, wird die aktuell laufende MiniMax-Suche abgebrochen und das Ergebnis der MiniMax-Suche mit der größten erreichten Tiefe zurückgegeben. [11]
In Abbildung 2.6 wird die Suche bei Tiefe 1 begonnen. Nach Abschluss der Suche für Tiefe 1 wird die Suche für Tiefe 2 gestartet, welche ebenfalls abgeschlossen wird. Während der MiniMax-Suche mit Tiefe 3 wird das Zeitlimit erreicht. Somit wird von der Suche kein Ergebnis zurückgegeben, da der komplette Suchbaum nicht erkundet werden konnte. *Iterative Deepening* gibt also das Ergebnis der Suche mit Tiefe 2 zurück.

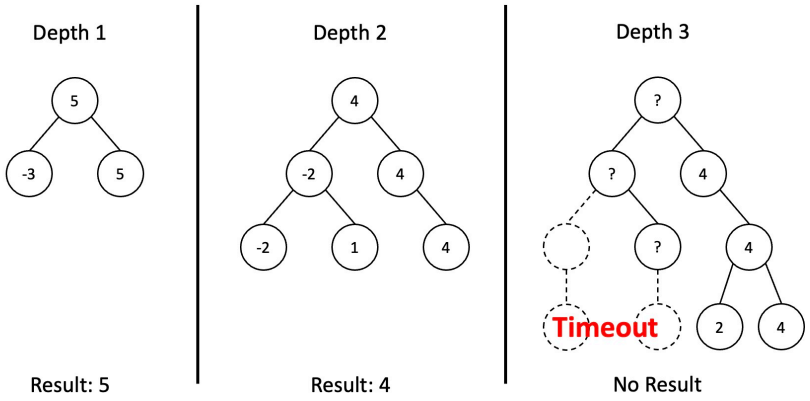


Abbildung 2.6: *Iterative Deepening* (Quelle: eigene Abbildung)

Iterative Deepening weist jedoch ein offensichtliches Problem auf: das wiederholte Erkunden gleicher Positionen. Wenn *Iterative Deepening* beispielsweise die Suchtiefe 5 erreicht, erscheint die verwendete Rechenzeit in den Tiefen 1 bis 4 wie verschwendet. Da jedoch die Anzahl der erkundeten Positionen exponentiell mit der Suchtiefe steigt, ist die benötigte Rechenzeit für die erreichte Tiefe T immer weit größer als der im Vergleich relativ kleine Zeitaufwand für die Tiefen 1 bis T-1. Der Zeitverlust durch *Iterative Deepening* statt direktes Aufruf der Tiefe T ist also weit geringer, als man zunächst erwarten würde. [12]
Zudem verwendet man Transpositionstabellen, sodass in den früheren Tiefen erkundete Positionen nicht erneut erkundet werden müssen, sondern das Ergebnis einfach aus der Transpositionstabelle abgerufen werden kann.
Ein großer Vorteil der Alpha-Beta-Suche ist die eingesparte Zeit durch die *Alpha-Beta-Cutoffs*. Diese sind maximal, wenn der beste Zug zuerst evaluiert wird. Evaluiert man also im *Iterative Deepening* zuerst den besten Zug aus der vorherigen Tiefe, ist es möglich, dass *Iterative Deepening* mehr *Cutoffs* erzeugt als ein normaler Aufruf der Alpha-Beta-Suche.
Ein weiterer Vorteil davon, den besten Zug aus der vorherigen Tiefe als erstes zu evaluieren, ist, dass man nicht mehr auf den alten besten Zug im Falle einer nicht vollständigen Suche zurückgreifen muss, da dieser bereits in der neuen höheren Tiefe erkundet wurde. Somit kann auch das Ergebnis der nicht vollendeten Suche akzeptiert werden. Obwohl *Iterative Deepening* eine einfache Zeitmanagement-Strategie ist, kann sie mithilfe dieser Strategien sogar schneller sein, als sofort die maximale Tiefe aufzurufen. [11]

2.2.8 Nullfenster-Suche

In der normalen Alpha-Beta-Suche werden die Parameter Alpha und Beta mit den Werten $-\infty$ und $+\infty$ initialisiert. Es wird also ein unendliches Suchfenster verwendet, um den kompletten Suchbaum zu erkunden. Alternativ dazu gibt es die Idee der Nullfenster-Suche. Dabei wird die Größe des Suchfensters extrem reduziert, nämlich zu einem leeren Suchintervall. Zum Beispiel im Bereich der ganzen Zahlen würde man Beta mit Alpha + 1 initialisieren, sodass das Intervall $[Alpha, Beta]$ keine Zahlen enthält. [19]

Durch das reduzierte Suchfenster kommt es während der Ausführung zu mehr *Cutoffs* als in der normalen Alpha-Beta-Suche. Dementsprechend ist eine Ausführung der Nullfenster-Suche schneller. Es werden jedoch auch weniger Informationen zurückgegeben.
Per Definition kann das Ergebnis der Nullfenster-Suche nicht zwischen Alpha und Beta liegen, sodass die Nullfenster-Suche nicht den optimalen MiniMax-Wert und damit auch keine *principal variation* zurückgeben kann.
Die zurückgegebenen Informationen der Nullfenster-Suche reichen jedoch aus, um den optimalen MiniMax-Wert einzugrenzen. Wenn die Suche tief fehlschlägt, das heißt, der zurückgegebene Wert kleiner als Beta ist, dient der Wert als obere Grenze für den optimalen MiniMax-Wert. Wenn die Suche hoch fehlschlägt, das heißt, der Wert ist größer gleich Beta, dann ist der Wert eine untere Grenze. [17]
Um also den optimalen MiniMax-Wert zu erhalten, sind innerhalb eines anderen Algorithmus wie MTD(f) mehrere Nullfenster-Suchen erforderlich, um den optimalen MiniMax-Wert einzugrenzen. In dem Abschnitt „MTD(f) Beispielausführung“ ist ein Beispiel dafür zu sehen, wie die Nullfenster-Suche mehrfach in MTD(F) ausgeführt wird.
Der erhöhte Aufwand durch das mehrfache Erkunden des Suchbaums wird in der Regel durch die Einsparungen mehr als ausgeglichen, sodass Algorithmen, die die Nullfenster-Suche nutzen, schneller als die normale Alpha-Beta-Suche sein können. [19]

2.2.9 MTD(F) Algorithmus

Der MTD(f)-Algorithmus, auch bekannt als Memory-enhanced Test Driver, ist eine einfache, aber auch schnelle und effiziente Variante von MiniMax. Dieser Algorithmus wurde von Aske Plaat, Jonathan Schaeffer, Wim Pijls und Arie de Bruin während ihrer Forschung am SSS*-Algorithmus entwickelt. [16]
MTD(f) verwendet mehrere Nullfenster-Suchen von Alpha-Beta, um den MiniMax-Wert einzugrenzen, da die Information einer einzelnen Nullfenster-Suche nicht ausreichen, um den genauen MiniMax-Wert zu ermitteln.
Der Algorithmus initialisiert einen Intervall mit einer oberen Schranke (*upperbound*) und einer unteren Schranke (*lowerbound*). Wenn die Nullfenster-Suche tief fehlschlägt, wird die obere Grenze nach dem zurückgegebenen Wert aktualisiert. Schlägt sie hoch fehl, dann wird die untere Grenze aktualisiert (wie im Abschnitt „Nullfenster-Suche“ erklärt). Wenn die Schranken aufeinandertreffen, wurde der optimale MiniMax-Wert gefunden, welcher dem Wert entspricht, an dem die Grenzen aufeinandertreffen.
Um sicherzustellen, dass dieser Ansatz funktioniert, benötigt man Alpha-Beta mit Transpositionstabellen, da dies den Overhead des erneuten Erkundens von Teilen des Suchbaums durch mehrfache Aufrufe von Alpha-Beta eliminiert. Auf diese Weise wird die Effizienz von MTD(f) gewährleistet.
Das „f“ in MTD(f) steht für den initialen *first guess*, den der Algorithmus benötigt. Dieser *first guess* dient als Startpunkt für die Suche mit Nullfenster-Suche. Je näher der *first guess* am optimalen MiniMax-Wert liegt, desto effizienter ist der Algorithmus. Im Idealfall ist der *first guess* gleich dem gesuchten MiniMax-Wert. Dann benötigt man nur zwei Nullfenster-Suchen: einen, um die obere Schranke zu aktualisieren, und einen weiteren, um die untere Schranke zu aktualisieren. [18]

Der Code für MTD(f) ist relativ kurz und simpel, wie in Listing 2.1 dargestellt. Zu Beginn werden die Grenzen initialisiert und der *first guess* übernommen. Solange sich die Grenzen nicht treffen,

werden Nullfenster-Suchen durchgeführt. Die Nullfenster-Suchen werden mithilfe des *first guess* oder des *guesses* der letzten Iteration gemacht. Für die Suchen muss nur Beta definiert werden, da Alpha immer eine Einheit kleiner als Beta ist. Das Ergebnis der Nullfenster-Suche wird genutzt, um die Grenzen zu aktualisieren. Nachdem sich die Grenzen getroffen haben, wird das letzte Ergebnis der Nullfenster-Suche zurückgegeben.

```
function MTDF(root : node_type; f : integer; d: integer) : integer;

    g := f;
    upperbound := +INFINITY;
    lowerbound := -INFINITY;
    repeat

        if g == lowerbound then beta := g + 1 else beta := g;
        g := AlphaBetaWithMemory(root, beta - 1, beta, d);
        if g < beta then upperbound := g else lowerbound := g;

    until lowerbound >= upperbound;
    return g;
```

Listing 2.1: MTD(f)-Algorithmus (Quelle: Mtd(f) algorithm, a minimax algorithm faster than negascout [18])

Im Normalfall würde man MTD(f) mittels *Iterative Deepening* einsetzten (siehe Listing 2.2). Man startet die Suche mit einem neuralen *first guess* von 0. In jeder weiteren Iteration ist es sinnvoll, das Ergebnis der vorherigen Iteration zu nutzen, da man so die beste Einschätzung des MiniMax-Wertes erhält.

```
function iterative_deepening(root : node_type) : integer;

    firstguess  := 0;
    for d  = 1 to MAX_SEARCH_DEPTH do

        firstguess := MTDF(root, firstguess, d);
        if times_up() then break;

return firstguess;
```

Listing 2.2: *Iterative Deepening* für MTD(f)-Algorithmus (Quelle: Mtd(f) algorithm, a minimax algorithm faster than negascout [18])

Diese Erklärung des MTD(f)-Algorithmus sowie die Beispielausführung und Abbildungen sind eine erweiterte Fassung einer Erklärung, welche ich bereits im Rahmen der Seminarleistung Algorithmen und Datenstrukturen verfasst habe.

MTD(f) Beispielausführung

Der MTD(f)-Algorithmus beginnt mit der Initialisierung der Schranken sowie der ersten Nullfenster-Suche basierend auf dem initialen Wert von f, wobei f gleich 1 ist:

$$g = f = 1 \tag{2.1}$$
$$upperbound = +\infty \tag{2.2}$$
$$lowerbound = -\infty \tag{2.3}$$
$$g < lowerbound \Rightarrow Beta = g = 1 \tag{2.4}$$

Daraus folgt der erste Nullfenster-Suche-Aufruf mit den Alpha-Beta-Intervall Grenzen:
$Beta = 1 \Rightarrow Alpha = Beta - 1 = 0$

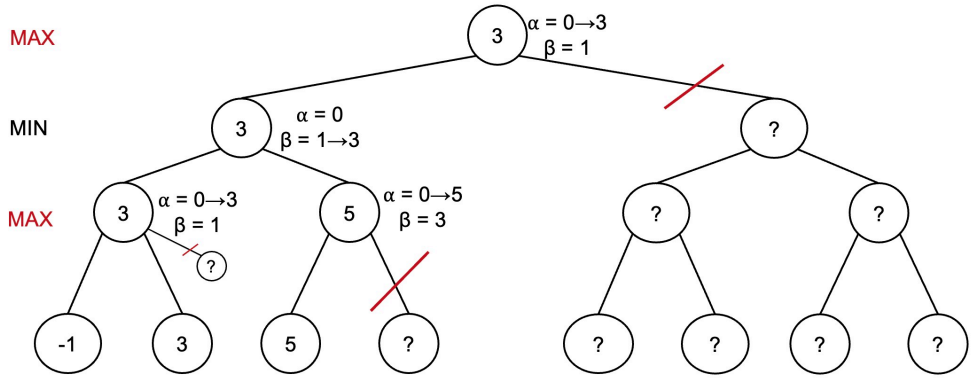


Abbildung 2.7: Erste Nullfenster-Suche von MTD(f) (Quelle: eigene Abbildung)

In Abbildung 2.8 fällt auf, dass durch einen *Cutoff* der komplette rechte Baum nicht erkundet wird. Dieser *Cutoff* würde in der „normalen“ Alpha-Beta-Suche nicht stattfinden. Die Nullfenster-Suche erkundet also wie erwartet nicht den kompletten Suchbaum und kann somit auch keine genaue Aussage über den optimalen MiniMax-Wert treffen. Der zurückgegebene Wert reicht jedoch aus, um Aussagen über die Grenzen des optimalen MiniMax-Werts zu machen:
Die Nullfenster-Suche hat den Wert 3 zurückgegeben, der größer als Beta ist. Die Suche ist also hoch fehlgeschlagen und somit kann die untere Grenze angepasst werden:

$$g = 3 > Beta = 1 \tag{2.5}$$
$$\Downarrow \tag{2.6}$$
$$lowerbound = 3 \tag{2.7}$$
$$g == lowerbound \Rightarrow Beta = g + 1 = 4 \tag{2.8}$$

Aus dem Ergebnis der vorherigen Nullfenster-Suche ergeben sich neue Grenzen und damit der zweite Nullfenster-Suche-Aufruf, der mit $Beta = 4$ durchgeführt wird:

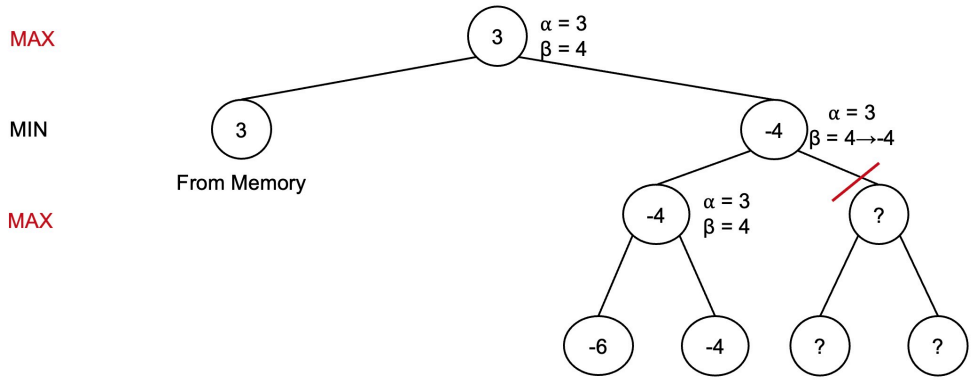


Abbildung 2.8: Zweite Nullfenster-Suche von MTD(f) (Quelle: eigene Abbildung)

In der zweiten Nullfenster-Suche ist die Verwendung von Transpositionstabellen essenziell, um die Effizienz von MTD(f) zu gewährleisten. Der komplette linke Teil des Suchbaums muss nicht erneut erkundet werden und kann aus der Tabelle ausgelesen werden. Mit dem rechten Teil des Suchbaums nun auch erkundet, ist der komplette Suchbaum ausreichend erkundet. Mithilfe des zuletzt zurückgegebenen Wertes können die Grenzen ein letztes Mal angepasst werden:
Aus der zweiten Nullfenster-Suche ergibt sich erneut der Wert 3. Dieses Mal ist der Wert jedoch kleiner als Beta, was bedeutet, dass die Suche tief fehlgeschlagen ist und die obere Grenze aktualisiert werden kann:

$$g = 3 < Beta = 4 \tag{2.9}$$
$$\Downarrow \tag{2.10}$$
$$upperbound = 3 \tag{2.11}$$
$$upperbound == lowerbound \Rightarrow Result = 3 \tag{2.12}$$

Die obere und untere Grenze haben sich getroffen. Somit ist der MTD(f)-Algorithmus nach zwei Nullfenster-Suchen auf den optimalen MiniMax-Wert konvergiert.

2.2.10 Implementation von Alpha-Beta in dieser Arbeit

Die Alpha-Beta-Suche kann nicht identisch für jedes Spiel eingesetzt werden. Die Evaluationsfunktion sowie die gewählten Verbesserungen hängen von dem Spiel ab, in dem die Alpha-Beta-Suche eingesetzt werden soll. Im Rahmen dieser Arbeit wird das Spiel „Catch the Lion“ betrachtet, daher die Details der Implementierung der Alpha-Beta-Suche:

Evaluationsfunktion

Die Evaluationsfunktion ist gleich der Evaluationsfunktion aus „MCTS-Minimax Hybrids with State Evaluations“ [5]. Die Evaluationsfunktion ist eine gewichtete Summe der Figurenwerte, wobei die Werte wie folgt sind:

$$\text{Sieg} = 1000 \tag{2.13}$$
$$\text{Huhn} = 3 \tag{2.14}$$
$$\text{Giraffe} = 5 \tag{2.15}$$
$$\text{Elephant} = 5 \tag{2.16}$$
$$\text{Henne} = 7 \tag{2.17}$$

Hierbei ist es unerheblich, ob die Figur auf dem Spielbrett steht oder in der Reserve des Spielers ist. Dazu werden die Werte des weißen maximierenden Spielers mit einem positiven Vorzeichen versehen und die des schwarzen minimierenden Spielers mit einem negativen Vorzeichen.

Zugreihenfolge

Das Ziel der Zugreihenfolge ist es, dass die vielversprechendsten Züge im Rahmen des Spiels „Catch The Lion“ zuerst evaluiert werden, um möglichst viele *alpha-beta-cutoffs* zu erzeugen. Die in dieser Arbeit angewandte Zugreihenfolge ist:

$$\text{Siegzüge} \rightarrow \text{Schlagzüge} \rightarrow \text{Figurenzüge} \rightarrow \text{Reservezüge} \tag{2.18}$$

Siegzüge: Im Rahmen von „Catch The Lion“ sind Siegzüge das Schlagen des gegnerischen Löwen oder das Bewegen des eigenen Löwen auf die letzte Reihe, also die Grundlinie des Gegners. Diese Züge haben die höchste Priorität, da sie das Spiel mit einem Sieg beenden und keine weiteren Züge evaluiert werden müssen.
Schlagzüge: Mit Schlagzügen sind Züge gemeint, in denen eine gegnerische Figur geschlagen wird. In „Catch The Lion“ haben Schlagzüge eine hohe Priorität, da sie den Evaluationswert dahingehend beeinflussen, dass die gegnerische Figurensumme sinkt und die eigene steigt, weil die geschlagene Figur in die Reserve gelegt wird. Zudem werden die Schlagzüge nach dem Prinzip des *Most Valuable Victim—Least Valuable Aggressor* priorisiert, da es bevorzugt ist, wertvollere Figuren mit weniger wertvollen Figuren zu schlagen. [5]
Figurenzüge: Figurenzüge sind das Bewegen einer Figur auf dem Spielbrett, ohne das eine gegnerische Figur geschlagen wird.
Reservezüge: Im Spiel „Catch The Lion“ sind Reservezüge das Platzieren einer Figur auf einem freien Feld auf dem Spielbrett. Diese Züge haben die niedrigste Priorität, da es oft pro Figur in Reserve recht viele Züge gibt mit nur minimalen unterschiedlichen Einfluss auf die Evaluationsfunktion. Um also möglichst wenig Zeit mit diesen Zügen zu verbrauchen, sollten zuerst alle anderen Züge evaluiert werden.

Transpositionstabellen

In dieser Arbeit genügte es, im Falle einer identischen Adresse in der Datenbank den alten Eintrag durch den neuen Eintrag zu ersetzen. Außerdem wird beim Zugriff der Hash des Eintrags mit dem Hash der aktuellen Stellung abgeglichen, um fehlerhafte Zugriffe zu vermeiden.

Iterative Deepening

Die Nutzung von *Iterative Deepening* im Rahmen dieser Arbeit ist recht rudimentär gehalten; es wird einfach iterativ die Tiefe erhöht, größtenteils ohne Verbesserungen.
Weder die MiniMax- noch die Alpha-Beta-Suche in dieser Arbeit nutzen *Iterative Deepening* mit Informationen aus der vorherigen Iteration, sodass die *principal variation* nicht in der nächsten Iteration beachtet wird.
Die Alpha-Beta-Suche mit Transpositionstabellen nutzt ebenfalls einfaches *Iterative Deepening*. Allerdings profitiert die Suche davon, dass Positionen, die in vorherigen Iterationen in die Transpositionstabelle eingetragen wurden, auch in späteren Iterationen abgerufen werden können.
Der MTD(f)-Algorithmus nutzt *Iterative Deepening* wie in Abschnitt MTD(F) Algorithmus beschrieben. Zusätzlich zur Nutzung der Transpositionstabellen kann die letzte Evaluation als *best guess* für die nächste Iteration verwendet werden.

2.3 Monte-Carlo-Tree-Search

Lange Zeit galt die Alpha-Beta-Suche als der Algorithmus für Zwei-Spieler-Nullsummenspiele. Trotz zahlreicher Verbesserungen des Suchverfahrens, erweist sich die Alpha-Beta-Suche in manchen Spielen wie „Go“ als nicht erfolgreich. [23]
Unter anderem wurde auch die Monte-Carlo-Methode in Kombination mit MiniMax von Bruce Abramson genutzt. Hierbei wurde die statische Evaluationsfunktion durch zufällige Spielsimulationen ersetzt. [15] Laut Abramson ist das erwartete outcome-Modell „präzise, genau, leicht abschätzbar, effizient berechenbar und domänenunabhängig“. [1]
Es dauerte jedoch einige Zeit, bis die von seinen Vorgängern inspirierte Monte-Carlo-Tree-Search entwickelt wurde und die Monte-Carlo-Methode breiten Erfolg sah. Im Gegensatz zu seinen Vorgängern erkundet die Monte-Carlo-Tree-Search nicht klassisch den Suchbaum gefolgt von Monte-Carlo-Evaluationen, sondern ist eine Best-First-Suche, die von den Ergebnissen der Monte-Carlo-Simulationen geleitet wird. In den folgenden Jahren erzielte die Monte-Carlo-Tree-Search große Erfolge im Bereich von Computer-Go [23]. Ein bemerkenswertes Ergebnis war, dass Google DeepMinds AlphaZero, eine Kombination aus Monte-Carlo-Tree-Search und Deep Learning, den Top-Go-Spieler Lee Sedol besiegte, was zuvor für nahezu unmöglich gehalten wurde.[2]

2.3.1 Funktionsweise

Der Monte-Carlo-Tree-Search Algorithmus besteht aus vier Schritten, die in einer Schleife wiederholt ausgeführt werden. Die unterschiedlichen Schritte sind *Selection*, *Expansion*, *Simulation* und *Backpropagation*, wobei jede Ausführung dieser vier Schritte der Simulation eines Spiels entspricht (siehe Abbildung 2.9). Währenddessen speichert jeder Knoten, wie oft er besucht wurde und wie viele Siege im Subbaum des Knotens gefunden wurden. Nach Ablauf einer vorgegebenen Zeit wird die Schleife abgebrochen und der beste berechnete Zug ausgewählt. [4]

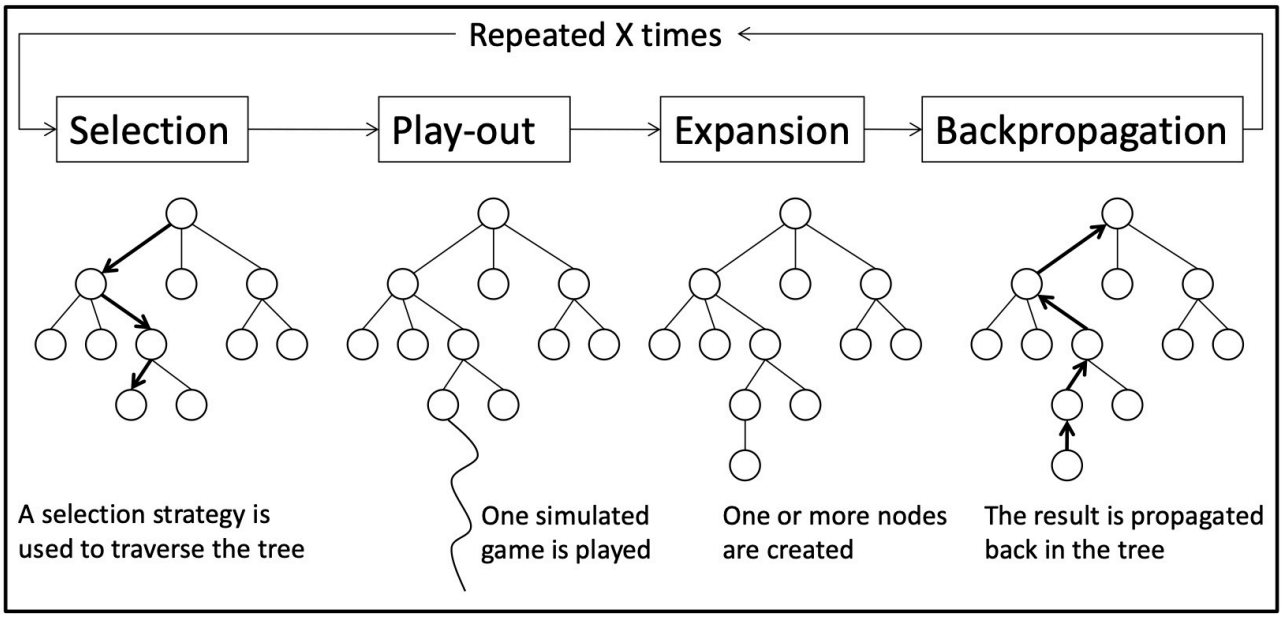


Abbildung 2.9: Monte-Carlo-Tree-Search-Schleife (Quelle: Monte-carlo tree search [24])

Selection

Die *Selection* beginnt im Wurzelknoten des Suchbaums. Mithilfe einer *Selection Policy* werden Kinderknoten ausgewählt, bis ein Blattknoten erreicht wird. Der Wurzelknoten ist der aktuelle Spielstand, für den die Monte-Carlo-Tree-Search ausgeführt wird. Als Blattknoten werden solche Knoten bezeichnet, für die noch kein Spiel simuliert wurde. [15]
Innerhalb dieser Arbeit wurde die populäre *UCT Policy (Upper Confidence bounds applied to Trees)* als *Selection Policy* gewählt.
Für Zwei-Spieler-Nullsummenspiele muss beachtet werden, dass die *Backpropagation* der NegaMax-Logik folgt. Das bedeutet, dass das Vorzeichen der Siegrate für die *UCT Policy* invertiert werden muss, damit der beste Zug für den Spieler, der im Knoten am Zug ist, selektiert wird.

Expansion

In der *Expansion*-Phase wird dem Knoten, der in der *Selection*-Phase ausgewählt wurde, ein neuer Kindknoten hinzugefügt; es sei denn, der Knoten ist terminal, also ein Sieg, eine Niederlage oder ein Unentschieden. Jeder Kindknoten repräsentiert einen möglichen Zug aus dem Spielstand des Knotens. [15]

Simulation

Während der *Simulation* wird ein Spiel für den neu expandierten Knoten simuliert. Dabei werden Züge zufällig oder pseudo-zufällig gewählt, bis das Spiel endet oder ein vorher festgelegtes Zuglimit erreicht wird . [23] Das Ergebnis der Simulation ist entweder -1 (Niederlage), 0 (Unentschieden) oder 1 (Sieg) für den Spieler, der am Zug ist. [15]

Backpropagation

Die *Backpropagation* gibt das Ergebnis der Simulation bis zum Wurzelknoten zurück, wobei die Siege und Besuche aller Knoten auf dem Weg entsprechend angepasst werden.
Im Fall von Zwei-Spieler-Nullsummenspielen wie „Catch The Lion“ ist zu beachten, dass die *Backpropagation* leicht anders funktioniert und einem ähnlichen Prinzip wie der NegaMax Variante von Alpha-Beta folgt. Da ein Sieg für den einen Spieler eine Niederlage für den anderen Spieler bedeutet, wird während der *Backpropagation* das Vorzeichen des Ergebnisses alterniert. So wird gewährleistet, dass während der *Selection* in jedem Knoten der beste Zug für den Spieler gewählt wird, der im Knoten auch am Zug ist. Dies entspricht einem realistischeren Abbild des Spiels. Anstatt das in jedem Knoten der beste Zug für den weißen Spieler gewählt wird, was eine unrealistische Repräsentation des tatsächlichen Spiels wäre.

Wahl des Zuges

Nach Ablauf der vorgegebenen Zeit wird die Schleife über die vier Schritte beendet. Aus den Kindknoten des Wurzelknotens muss jedoch noch der beste Zug gewählt werden. Es ist möglich, den Knoten zu wählen, der am häufigsten besucht wurde, den Knoten mit den meisten Siegen, oder eine Kombination der beiden, wie die Siegrate. In der Praxis macht es keinen großen Unterschied, welche Variante man wählt, solange ausreichend Simulationen durchgeführt wurden. [23]

2.3.2 *UCT Policy (Upper Confidence bounds applied to Trees)*

Die *UCT Policy* behandelt die *Selection* von Monte-Carlo-Tree-Search als *multi-armed bandit problem*. Die Entscheidung an jedem Knoten ist also zwischen *exploration* und *exploitation*. Mit *exploration* ist die Erkundung der anderen Spielautomaten gemeint, um mehr Informationen über ihre Gewinnchance zu erhalten. Wohingegen mit *exploitation* das Weiterspielen am Spielautomaten mit der am höchsten erwarteten Auszahlung gemeint ist. [21] Im Kontext von Monte-Carlo-Tree-Search bedeutet das die Entscheidung zwischen dem aktuelle vielversprechendsten Knoten (*exploitation*) und den Knoten, welche noch nicht ausreichend erkundet wurden (*exploration*).

Der UCT-Wertes eines Knotens i berechnet sich mit folgender Formel:

$$UCT_{\mathrm{i}} = X_{\mathrm{i}} + C * \sqrt{\frac{ln(N_{\mathrm{i}})}{n_{\mathrm{i}}}} \tag{2.19}$$

$$X_{\mathrm{i}} = \text{Siegrate des Knoten} \tag{2.20}$$

$$C = \text{Konstante zur Kontrolle der Erkundung} \tag{2.21}$$

$$n_{\mathrm{i}} = \text{Besuche des Knoten i} \tag{2.22}$$

$$N_{\mathrm{i}} = \text{Besuche des Elternknoten von Knoten i} \tag{2.23}$$

Der vordere Teil, also die Siegrate, repräsentiert die *exploitation*, da dieser hoch ist für Knoten mit hoher Siegrate. Der hintere Teil repräsentiert *exploration*, weil der Wert groß ist für Knoten mit wenig Simulationen ist. Die Konstante C kontrolliert den Einluss des hinteren Teils der Formel und damit die Balance zwischen *exploitation* und *exploration*. C wird oft als $\sqrt{2}$ gewählt nach der UCB1 Formel. [21]
Mit unendlich Zeit und Speicher konvergiert Monte-Carlo-Tree-Search mit *UCT Policy* zu MiniMax. [21]

2.3.3 Handsimulation von MCTS

Die Monte-Carlo-Tree-Search für Zwei-Spieler-Nullsummenspiele wie „Catch The Lion“ unterscheidet sich durch die NegaMax-Logik in *Selection* und *Backpropagation* von der generellen Monte-Carlo-Tree-Search. Die Handsimulation umfasst eine Iteration der Monte-Carlo-Tree-Search in dem Zwei-Spieler-Nullsummenspiel „Catch The Lion“.

Selection

Während der *Selection* wird entlang des Suchbaums der beste Knoten mit der *UCT Policy* gewählt. Hierbei muss das Vorzeichen der Siegrate in der Berechnung von UCT invertiert werden. In der Grafik 2.10 ist der Knoten (3/5) zu erkennen. In dem Knoten führen drei der Simulationen zum Sieg für den Spieler, welcher im Knoten am Zug ist. Für den Spieler, der im Wurzelknoten am Zug ist, bedeutet dies jedoch drei Niederlagen. Daher wird während der *Selection* aus dem Wurzelknoten das Vorzeichen in der Berechnung von UCT invertiert. Der Knoten (-1/2) maximiert die *UCT Policy* (siehe Gleichung 2.25), da er eine vielversprechende Siegrate hat und noch nicht viel erkundet wurde. Es wird also der Knoten (-1/2) selektiert, gefolgt von der *Selection* des (1/1) Knotens.

$$0.092 = -3/5 + \sqrt{\frac{ln(11)}{5}} \tag{2.24}$$

$$1.594 = 1/2 + \sqrt{\frac{ln(11)}{2}} \tag{2.25}$$

$$1.024 = 1/4 + \sqrt{\frac{ln(11)}{4}} \tag{2.26}$$

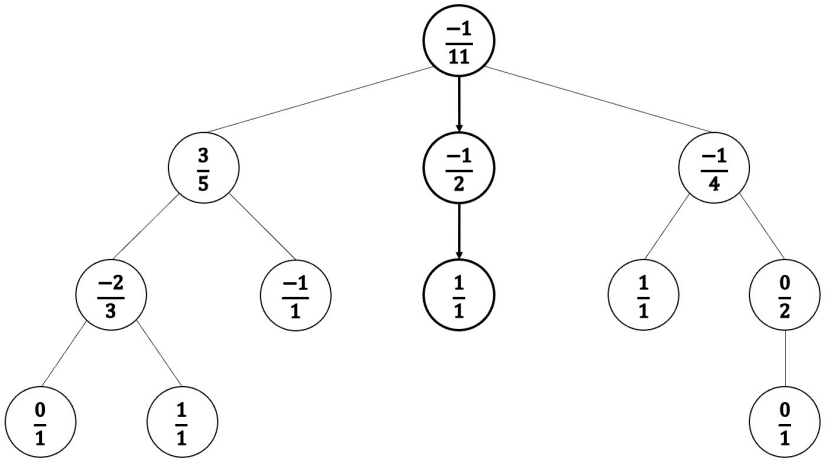


Abbildung 2.10: Monte-Carlo-Tree-Search Selection (Quelle: eigene Abbildung)

Expansion

Während der *Expansion* wird ein neuer Knoten an der durch die *Selection* gewählten Stelle im Suchbaum hinzugefügt (siehe Abbildung 2.11).

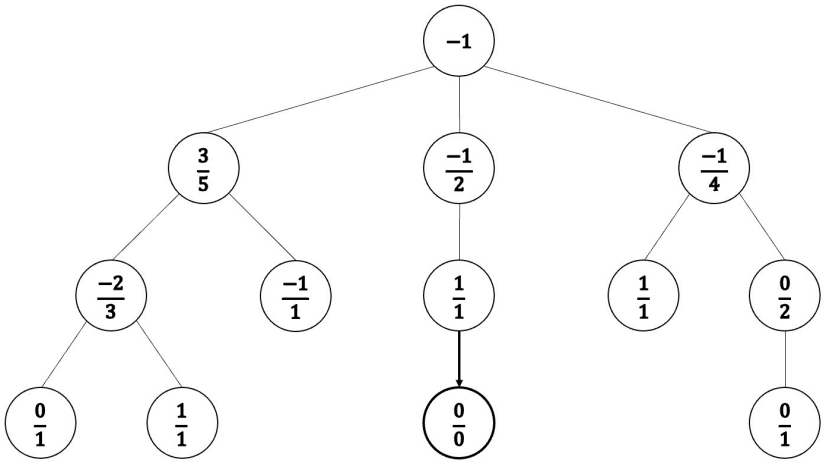


Abbildung 2.11: Monte-Carlo-Tree-Search Expansion (Quelle: eigene Abbildung)

Simulation

Für den neu hinzugefügten Knoten während der *Expansion* wird eine Simulation durchgeführt. Das Ergebnis der Simulation bezieht sich auf den Spieler, der im neuen Knoten am Zug ist. In Abbildung 2.12 ergibt die Simulation eine Niederlage für den neuen Knoten.

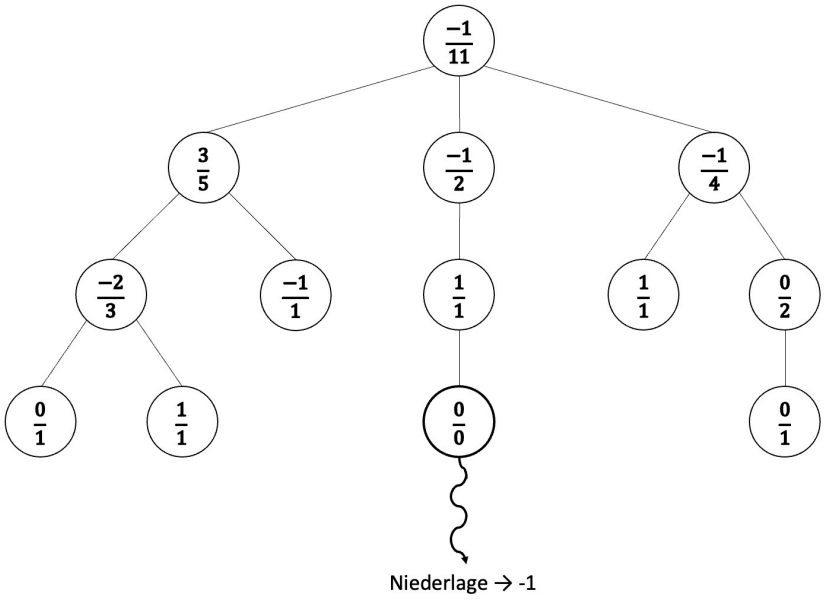


Abbildung 2.12: Monte-Carlo-Tree-Search Simulation (Quelle: eigene Abbildung)

Backpropagation

Zuletzt wird das Ergebnis der *Simulation* durch den Baum bis zum Wurzelknoten zurückpropagiert. Da in diesem Beispiel Monte-Carlo-Tree-Search für ein Zwei-Spieler-Nullsummenspiel eingesetzt wird, ändert sich das Vorzeichen des zurückgegebenen Wertes für jeden Knoten. Wie in Abbildung 2.13 zu erkennen ist, bedeutet die Niederlage im neuen Knoten ein Sieg für den Elternknoten. Außerdem erhöht sich die Anzahl der Besuche entlang des Weges zum Wurzelknoten.

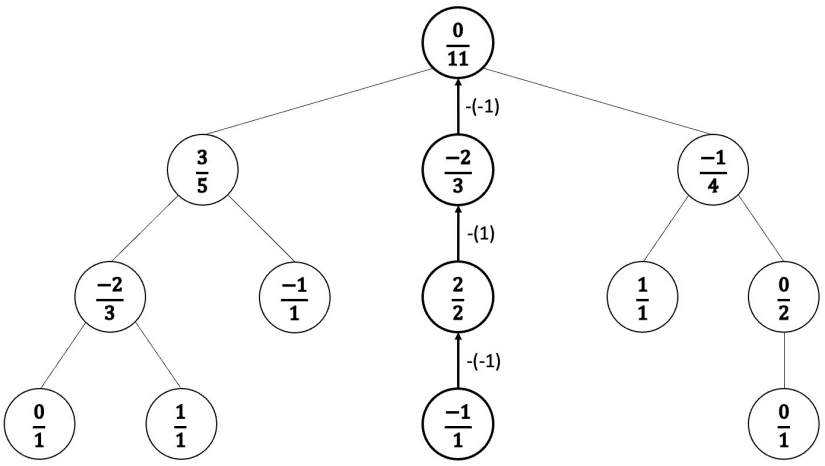


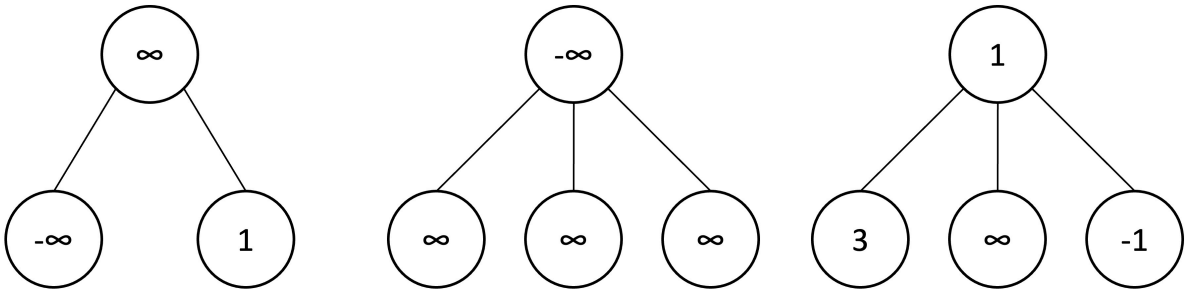
Abbildung 2.13: Monte-Carlo-Tree-Search Backpropagation (Quelle: eigene Abbildung)

Die Ausführung von *Selection*, *Expansion*, *Simulation* und *Backpropagation* wird solange in einer Schleife wiederholt, bis die vorgegebene Zeit abgelaufen ist. Zum Ende wird die Schleife unterbrochen und auf Basis einer Richtlinie, wie zum Beispiel der höchsten Anzahl an Besuchen, der beste Zug gewählt.

2.3.4 MCTS-Solver

Monte-Carlo-Tree-Search ist mit *UCT* in der Lage den besten Zug zu finden, MCTS kann jedoch in Spielen mit *sudden death*, wie zum Beispiel „Lines of Action “ oder „Catch the Lion “ auf Probleme stoßen, da der Algorithmus nicht in der Lage ist, spieltheoretischen Werte zu beweisen und so schnell den optimalen Zug zu finden.[23]
Spieltheoretische Werte im Kontext von „Catch the Lion“ meinen die Bewertung einer Stellung als gewonnen oder verloren für den Spieler, der am Zug ist.
Zum Beispiel könnte es eine Spielsituation geben, in der nach 5 Sekunden Suche ein Zug mit einer Simulationsgewinnrate von 67% gewählt wird, jedoch führt dieser Zug zu einer erzwungenen Niederlage nach 8 Zügen. Der optimale Zug hingegen weist nur eine Gewinnrate von 48% auf, führt aber zu einem erzwungenen Sieg in 7 Zügen. Erst wenn man Monte-Carlo-Tree-Search 60 Sekunden suchen lässt, findet der Algorithmus den optimalen Zug. Im Vergleich dazu benötigt Alpha-Beta weniger als eine Sekunde, um den optimalen Zug zu finden, der zu einem garantierten Sieg führt. [23]
Um dieses Problem zu lösen, entwickelten Mark H.M. Winands, Yngvi Björnsson und Jahn-Takeshi Saito eine neue MCTS-Variante, den MCTS-Solver. Dieser ist in der Lage, spieltheoretische Werte durch den Suchbaum zu propagieren, um so *proven wins* und *proven losses* zu finden.[23]
Ein Knoten wird als *proven win* bewertet, wenn für den Spieler, der in dem Knoten am Zug ist, garantiert ist, dass er den Sieg erringen wird, sobald er die Stellung im Knoten erreicht. Analog dazu ist ein Knoten ein *proven loss*, wenn die Niederlage für den Spieler garantiert ist.

Backpropagation



(a) proven win (b) proven loss (c) loss

Abbildung 2.14: MCTS-Solver-*Backpropagation* (Quelle: eigene Abbildung)

Um spieltheoretische Werte zurückzupropagieren, verwendet der MCTS-Solver nicht nur die Werte 1, 0 und -1 (für Sieg, Unentschieden und Niederlage), sondern auch die Werte $-\infty$ und $+\infty$. Wenn der MCTS-Solver während der Suche auf eine Endstellung trifft, wird der entsprechende Knoten im Falle einer Niederlage (*proven loss*), wenn beispielsweise der Löwe geschlagen wurde, mit $-\infty$ bewertet oder im Falle eines Sieges (*proven win*), zum Beispiel beim Bewegen des Löwen auf die letzte Reihe, mit $+\infty$. Die Bewertung erfolgt aus der Perspektive des Knotens; wenn also der Spieler, der im Knoten am Zug ist, gewinnt, handelt es sich um einen *proven win*. [23]
Die *Backpropagation* der spieltheoretischen Werte folgt der gleichen Logik wie die NegaMax-Variante der Alpha-Beta-Suche. Wenn ein Kindknoten den Wert $-\infty$ zurückgibt, handelt es sich um einen *proven loss* für den Kindknoten. Gemäß NegaMax bedeutet dies, dass der Elternknoten den Wert $+\infty$,

also einen *proven win*, erhält. Wenn zum Beispiel Weiß einen Zug hat, nach dem Schwarz nur Züge hat, die zur Niederlage führen, dann führt dieser Zug für Weiß zum Sieg. Es genügt also, wenn ein Kindknoten ein *proven loss* ist, um den *proven win* für den Elternknoten zu beweisen (siehe Abbildung 2.14a).[23, inspiriert]
Gibt der Kindknoten den Wert $+\infty$ zurück, also einen *proven win* für den Kindknoten, dann muss für alle anderen Kindknoten geprüft werden, ob sie ebenfalls zu einem *proven win* führen. Führen alle Kindknoten zu einem *proven win*, so lässt sich für den Elternknoten schlussfolgern, dass es sich um einen *proven loss* handelt und $-\infty$ kann zurückpropagiert werden. Andernfalls wird nur eine normale Simulationsniederlage zurückpropagiert. Denn nur wenn alle Züge von Weiß zum garantierten Sieg von Schwarz führen, ist die Niederlage für Weiß garantiert (siehe Abbildung 2.14b und 2.14c). [23, inspiriert]

Wahl des Zuges

Im Gegensatz zur normalen Monte-Carlo-Tree-Search reicht es nicht aus, den Knoten mit den meisten Besuchen oder Siegen zu wählen. Durch die *Backpropagation* der spieltheoretischen Werte von MCTS-Solver kann die Bewertung eines Knotens plötzlich steigen oder sinken. Die Wahl des besten Knotens muss also leicht anders erfolgen.[23]
In dem Fall, dass ein *proven win* bis zum Wurzelknoten propagiert wurde, kann die Suche sofort beendet werden. Man wählt natürlich den Kindknoten, der zum garantierten Sieg führt. Dies könnte zum Beispiel der Fall in Mate-in-X-Positionen sein, in denen der garantierte Sieg nur noch X Züge entfernt ist. [23]
Ist das nicht der Fall, wurde von Mark H. M.Winands, Yngvi Björnsson und Jahn-Takeshi Saito für den „Monte-Carlo Tree Search Solver“ die Methode namens *secure child* gewählt.[23] Der *secure child* ist der Knoten, welcher eine untere Vertrauensgrenze maximiert. Der *secure child* Knoten maximiert die folgende Funktion [7]:

$$v + \frac{A}{\sqrt{n}} \tag{2.27}$$

$$v = \text{Siegrate des Knoten} \tag{2.28}$$

$$A = \text{Parameter (Hier gleich 1)} \tag{2.29}$$

$$n = \text{Besuche des Knoten} \tag{2.30}$$

Pseudocode für MCTS-Solver

Der Pseudocode ist eine angepasste Version des Pseudocodes aus dem Monte-Carlo-Tree-Search Solver von Mark H.M. Winands, Yngvi Björnsson und Jahn-Takeshi Saito.[23]
`MCTS_Solver_Run` ruft `MCTS_Solver` solange auf, bis nach Ablauf der Schleife mittels `secure_child(root)` der beste Zug gewählt wird.
Die Funktion `MCTS_Solver` enthält alle vier Schritte: *Selection*, *Expansion*, *Simulation* und *Backpropagation*. Zu Beginn der Funktion wird für jeden Knoten beim ersten Besuch geprüft, ob es sich um einen *proven win* oder *proven loss* handelt. Hierfür überprüft die Funktion `playerToMoveWins(N)`, ob der Spieler, welcher im Knoten N am Zug ist, einen Zug hat, der zum Sieg führt. Die Funktion `playerToMoveLoses(N)` funktioniert äquivalent.
Die Funktion `Selection(N)` wählt entweder den besten Kindknoten nach der *UCT Policy* oder einen neu expandierten Kindknoten, wenn noch nicht alle möglichen Kindknoten von N mindestens einmal besucht wurden. Danach wird entweder eine Simulation für den neu expandierten Kindknoten durchgeführt oder `MCTS_Solver(best_child)` rekursiv ausgeführt, bis ein Blattknoten erreicht wird. In der `Simulation(best_child)` Funktion wird für den übergebenen Knoten ein Spiel bis zum Ende simuliert, wobei die Züge zufällig gewählt werden. Wenn die Simulation zu einem Sieg für den Spieler führt, der im Knoten am Zug ist, wird eine 1 zurückgegeben. Im Falle einer Niederlage wird eine -1 zurückgegeben und bei einem Unentschieden eine 0. Hier wird auch die NegaMax-Charakteristik von MCTS-Solver klar, denn die zurückgegebenen Werte von `Simulation(best_child)`, `MCTS_Solver(best_child)` und `best_child.score` werden alle negiert, da ein Sieg für den Kindknoten eine Niederlage für den Elternknoten bedeutet.
Im letzten Teil des Pseudocodes findet die Backpropagation von MCTS-Solver statt. Die Funktion `update_score(R)` bedeutet lediglich, dass der Wert R zum Wert des Knotens hinzuaddiert wird.

```
def MCTS_Solver_Run(state: Board, whiteTurn: bool, time: int):
    root = MCTS_Node.MCTS_Node(state, whiteTurn)
    while time left do:
        MCTS_Solver(root)
    return secure_child(root)

def MCTS_Solver(N: MCTS Node):
    if first visit of Node:
        if playerToMoveWins(N):
            N.score = ∞
            return ∞
        else if playerToMoveLoses(N):
            N.score = -∞
            return -∞
    best_child = Selection(N)
    N.visits = N.visits + 1
    if best_child.score != ∞ and
       best_child.score != -∞:
        if best_child.visits == 0:
            R = -Simulation(best_child)
            best_child.update_score(-R)
            best_child.visits++
            N.add_child(best_child)
            N.update_score(R)
            return R
        else:
            R = -MCTS_Solver(best_child)
    else:
        R = -best_child.score

    if R == ∞:
        N.score = ∞
        return N.score
    elif R == -∞:
        for child in N.children:
            if -child.score != R:
                R = -1
                N.update_score(R)
                return R
        N.score = -∞
        return N.score
    else:
        N.update_score(R)
        return R
```

Listing 2.3: MCTS-Solver Pseudocode (Quelle: Monte-carlo tree search solver [23])

2.3.5 MCTS gegen MCTS-Solver

Der MCTS-Solver-Algorithmus wurde von Mark H.M. Winands, Yngvi Björnsson und Jahn-Takeshi Saito mit dem Standard-Monte-Carlo-Tree-Search-Algorithmus im Spiel „Line of Action“ verglichen. Es wurden 1000 Spiele simuliert, in denen beide Algorithmen 5 Sekunden pro Zug hatten. Davon gewann der MCTS-Solver 65%, wie in Abbildung 2.15 zu sehen ist. Somit ist der MCTS-Solver eine Verbesserung des Monte-Carlo-Tree-Search-Algorithmus.[23]

	Score	Win %	Winning ratio
MCTS-Solver vs. MCTS	646.5 - 353.5	65%	1.83

Abbildung 2.15: MCTS gegen MCTS-Solver in LOA (Quelle: Monte-carlo tree search solver [23])

Außerdem wurden MCTS und MCTS-Solver auch mit drei Versionen von MIA, einem Alpha-Beta-Algorithmus für das Spiel „Line of Action“, verglichen. Die Versionen unterscheiden sich durch ihre Eval-Funktion. [23]
In Abbildung 2.16 ist zu erkennen, dass es gegen MIA 2000 beiden MCTS-Algorithmen gelingt, mehr als 50% der Spiele zu gewinnen. Jedoch gewinnt MCTS-Solver mehr Spiele. Gegen MIA 2002 ist nur noch MCTS-Solver in der Lage, mehr als 50% der Spiele zu gewinnen. Gegen MIA 2006 ist keiner der MCTS-Algorithmen in der Lage, mehr als 50% der Spiele zu gewinnen, dennoch ist MCTS-Solver in der Lage, mehr Spiele als MCTS zu gewinnen.
Es lässt sich also sagen, dass MCTS-Solver Line of Action besser spielt als MCTS, jedoch noch nicht so gut wie moderne Alpha-Beta-Algorithmen.[23]

Evaluator	MIA 2000	MIA 2002	MIA 2006
MCTS	585.5	394.0	69.5
MCTS-Solver	692.0	543.5	115.5

Abbildung 2.16: MCTS und MCTS-Solver gegen MIA in LOA (Quelle: Monte-carlo tree search solver [23])

2.4 MCTS-MiniMax Hybride

Auch wenn MCTS-Solver Monte-Carlo-Tree-Search deutlich in Spielen mit *sudden deaths* verbessert, können MCTS-Zufallssimulationen wichtige Züge übersehen oder die Bedeutung einer Endstellung während der Backpropagation unterschätzen. Um die Spielstärke von Monte-Carlo-Tree-Search, besonders in Spielen mit hoher Dichte von *sudden deaths*, zu verbessern, schlugen Hendrik Baier und Mark H. M. Winands in ihrem Paper „MCTS-Minimax Hybrids“ ein Hybrid aus MiniMax und MCTS vor. [4]
Dabei wird MiniMax in den Monte-Carlo-Tree-Search Algorithmus integriert, um das strategische Können von MCTS mit der taktischen Finesse von MiniMax zu kombinieren. MCTS zeichnet sich durch die Fähigkeit aus, strategisch gute Entscheidungen zu treffen, da die Suche auf der Simulation vieler Spiele bis zum Ende beruht. Jedoch kann MCTS taktisch versagen, indem der Algorithmus *sudden deaths* übersieht. Hierbei fungiert MiniMax im Gegensatz dazu durch starkes taktisches Spiel über die Evaluierung aller möglichen Züge bis zu einer Tiefe von X. MiniMax ist jedoch strategisch schwach, da die Weitsicht über diese Tiefe X hinaus fehlt. Somit könnte ein Hybrid aus MCTS und MiniMax einen universell nützlicheren Suchalgorithmus darstellen. [4]
Hierfür wird die MiniMax-Suche in jeweils die *Simulation*, *Selection* oder *Backpropagation* Phase des MCTS-Solver-Algorithmus integriert, was drei verschiedene MCTS-MiniMax-Hybride ergibt: *MCTS-MR, Monte Carlo Tree Search with MiniMax-Rollout; MCTS-MS, Monte Carlo Tree Search with MiniMax-Selection; MCTS-MB, Monte Carlo Tree Search with MiniMax-Backpropagation.* [4]
Jeder der Hybridalgorithmen in dieser Arbeit nutzt MiniMax mit *alpha-beta-pruning* sowie einer simplen Evaluationsfunktion, welche nur Sieg oder Niederlage zurückgibt.

2.4.1 MiniMax in der *Simulation*-Phase, MCTS-MR

Die Integration von MiniMax in die *Simulation*-Phase ist durchaus einfach und sinnvoll. Auch wenn MCTS mit zufällig gewählten Zügen während der Simulation zum optimalen Zug konvergiert [21], verbessert eine präzisere Simulation die Spielstärke von MCTS [4].
Eine Simulation mittels MiniMax funktioniert wie zu erwarten, sodass während der Simulation der Zug immer mittels einer MiniMax-Suche mit fester Tiefe gewählt wird. Für diese Arbeit wurde eine simple Evaluationsfunktion gewählt, welche nur Sieg und Niederlage betrachtet.
Durch die Verwendung der Alpha-Beta-Suche in der *Simulation*-Phase anstelle von Zufallssimulationen weist MCTS-MR im Gegensatz zur Monte-Carlo-Tree-Search kein Zufallselement auf und ist daher deterministisch. Da in dieser Arbeit die Evaluationsfunktion nur zwischen einem Sieg (1) und einer Niederlage (-1) unterscheidet, wird oft der Wert 0 zurückgegeben, wenn weder ein Sieg noch eine Niederlage erkannt wurde. In solchen Fällen wird weiterhin zufällig zwischen den Zügen mit derselben Bewertung in der Alpha-Beta-Suche ausgewählt, um dem Zufallsaspekt der normalen Monte-Carlo-Tree-Search zu folgen. Daher bleibt MCTS-MR in dieser Arbeit weiterhin nicht deterministisch.
In Abbildung 2.17 ist zu sehen, dass die MiniMax-Suche einen Sieg für den Gegner findet (Zug A) und deshalb den Zug B wählt, gefolgt von einer weiteren MiniMax-Suche.
Diese Strategie verbessert die Monte-Carlo-Tree-Search, indem sie das Umgehen von *sudden deaths* während der Simulation ermöglicht [4].

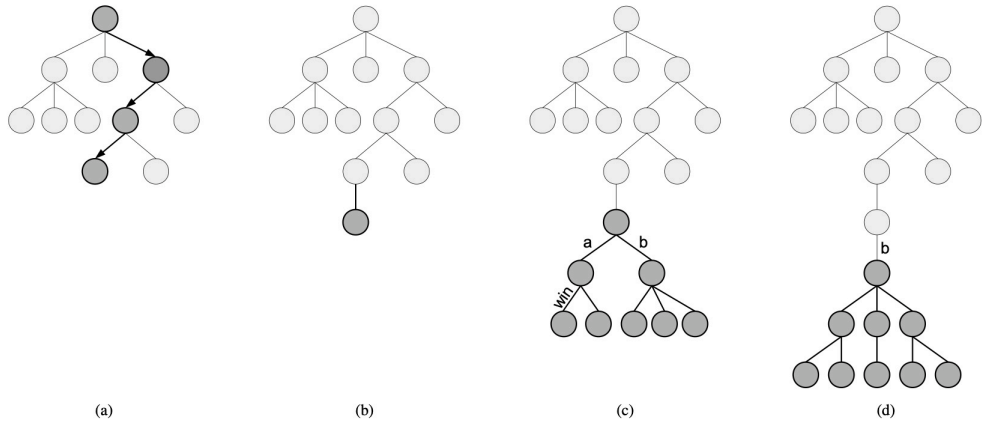


Abbildung 2.17: MiniMax in der *Simulation*-Phase
(a) *Selection*-Phase, (b) *Expansion*-Phase, (c) Bei der MiniMax-Simulation mit einer Suchtiefe von 2 für den neuen Knoten wird ein Sieg für den Gegner nach Zug a gefunden. Daher wird Zug b gespielt. (d) Es wird eine weitere MiniMax-Suche gestartet. Da weder ein Sieg noch eine Niederlage gefunden wurde, wird der Zug zufällig gewählt. (Quelle: Mcts-minimax hybrids [4])

Pseudocode MCTS-MR

Der Pseudocode von MCTS-MR ist beinahe identisch zu dem von MCTS-Solver, es kommt lediglich ein neuer Parameter dazu, *Depth*, der die Tiefe der MiniMax-Suche bestimmt. In Zeile 7 wird `MiniMax_Simulation(best_child, Depth)` aufgerufen, um ein Spiel mittels MiniMax zu simulieren.

```
1  def MCTS_MR(N: MCTS Node, Depth: Integer):
2      Selection wie im MCTS-Solver
3
4      if best_child.score != ∞ and
5         best_child.score != -∞:
6          if best_child.visits == 0:
7              R = -MiniMax_Simulation(best_child, Depth)
8              best_child.update_score(-R)
9              best_child.visits++
10             N.add_child(best_child)
11             N.update_score(R)
12             return R
13         else:
14             R = -MCTS_MR(best_child, Depth)
15     else:
16         R = -best_child.score
17
18     Backpropagation wie im MCTS-Solver
```

Listing 2.4: MCTS-MR Pseudocode (Quelle: Erweiterung von Monte-carlo tree search solver [23])

2.4.2 MiniMax in der *Selection*-Phase, MCTS-MS

Die Idee, MiniMax-Suche während der *Selection*-Phase zu nutzen, besteht darin, schon während der *Selection proven wins* und *proven losses* zu finden. Dies funktioniert so, dass während der *Selection* eine MiniMax-Suche ausgeführt wird, wenn ein Knoten das notwendige Kriterium erfüllt. Die Evaluationsfunktion gibt hier nur Sieg oder Niederlage zurück, somit reicht die Information aus, um einen *proven win* oder *proven loss* zu bestimmen und diesen zurückzupropagieren. Andernfalls wird die *Selection* normal fortgesetzt. Als Kriterium reicht die Anzahl der Besuche des Knotens. Erreichen diese ein *Visit Threshold*, wird MiniMax ausgeführt. Andere Kriterien sind auch möglich, werden aber nicht in dieser Arbeit verwendet [4].
In Abbildung 2.18 wird die MiniMax-Suche während der *Selection*-Phase eingesetzt. Wird so ein Sieg oder eine Niederlage bewiesen, wird diese zurückpropagiert. Andernfalls wird die *Selection* fortgesetzt.
MCTS-MS verbessert die Monte-Carlo-Tree-Search, indem schon während der *Selection* der Suchbaum durch das Finden von *proven wins* und das Ausweichen von *proven loss* geleitet wird [4].

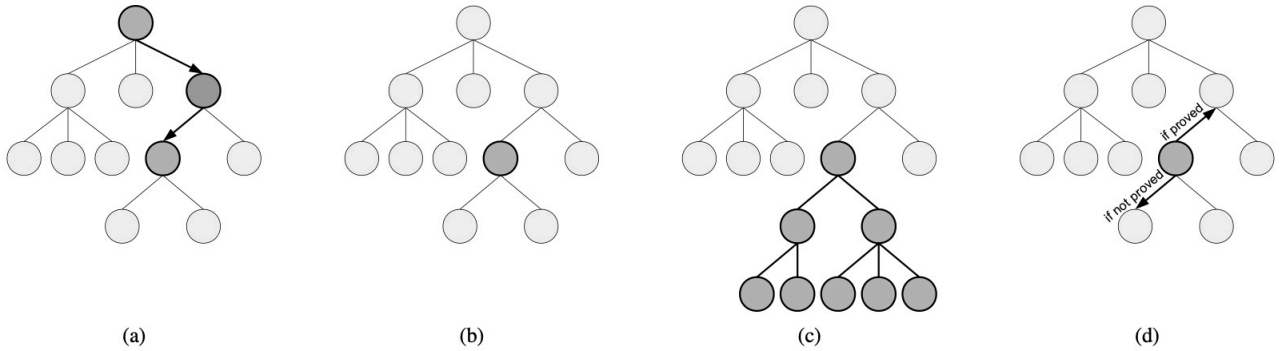


Abbildung 2.18: MiniMax in der *Selection*-Phase
(a) Start der *Selection*-Phase, (b) Während der *Selection* wird ein Knoten gefunden, der das Kriterium des *Visit Thresholds* erfüllt. (c) Für diesen Knoten wird eine MiniMax-Suche durchgeführt. (d) Wenn für den Knoten ein Sieg oder eine Niederlage bewiesen werden kann, werden diese Informationen zurückpropagiert. Andernfalls wird die *Selection* wie gewohnt fortgesetzt. (Quelle: Mcts-minimax hybrids [4])

Pseudocode MCTS-MS

MCTS-MS Code unterscheidet sich von MCTS-Solver durch die vor der *Selection* hinzugefügte MiniMax-Suche in den Zeilen 11-18. Hier wird, wenn das Besuchskriterium erfüllt wird, eine MiniMax-Suche ausgeführt, um nach einem *proven win* oder *proven loss* zu suchen.
Aufgrund des Besuchskriteriums benötigt MCTS-MS auch einen neuen Parameter `Visit Threshold`, sowie die Tiefe für die MiniMax-Suche `Depth`.

```
1  def MCTS_MS(N: MCTS Node, Depth: Integer, Visit Threshold: Integer):
2      if first visit of Node:
3          if playerToMoveWins(N):
4              N.score = ∞
5              return ∞
6          else if playerToMoveLoses(N):
7              N.score = −∞
8              return −∞
9
10     if Visit Threshold is met:
11         result = AlphaBeta(Depth)
12         if AlphaBeta proves Win for Node:
13             N.score =∞
14             return ∞
15         else if AlphaBeta proves Loss for Node:
16             N.score = − ∞
17             return − ∞
18
19     best_child = Selection(N)
20     N.visits = N.visits + 1
21
22     Simulation und Backpropagation wie im MCTS−Solver
```

Listing 2.5: MCTS-MS Pseudocode (Quelle: Erweiterung von Monte-carlo tree search solver [23])

2.4.3 MiniMax in der *Backpropagation*-Phase, MCTS-MB

Die MiniMax-Suche während der *Backpropagation* dient als *Fallback* für den MCTS-Solver. Wenn ein *proven loss* zurückpropagiert werden muss, muss sobald ein Kindknoten kein *proven loss* ist, zur normalen *Backpropagation* von Niederlagen gewechselt werden. In diesem Fall wendet MCTS-MB die MiniMax-Suche an; es wird also aktiv nach einem *proven loss* gesucht, anstatt zu hoffen, dass in späteren Simulationen MCTS-Solver diesen findet. Gelingt es MiniMax, die Position zu beweisen, kann weiterhin ein *proven loss* zurückpropagiert werden, anstelle einer normalen Niederlage [4].
In Abbildung 2.19 ist zu sehen, dass die *Backpropagation* eines *proven loss* auf einen Knoten trifft, von welchem ein Kindknoten noch nicht als *proven loss* bewiesen wurde. Mit Hilfe der MiniMax-Suche kann jedoch für den Kindknoten der *proven loss* bewiesen werden. Dementsprechend kann nach NegaMax ein *proven win* für den Elternknoten zurückpropagiert werden.
MCTS-Solver wird durch MiniMax in der *Backpropagation*-Phase dahingehend verbessert, dass die Suche hilft, *proven win* und *proven loss* zu erkennen, und somit Züge von weiteren Simulationen ausgeschlossen werden können. Da MiniMax auch nur im Kontakt mit Endstellungen ausgeführt wird, ist MCTS-MB in der Lage, Rechenzeit in Positionen mit wenig oder keinen Endstellungen zu sparen [4].

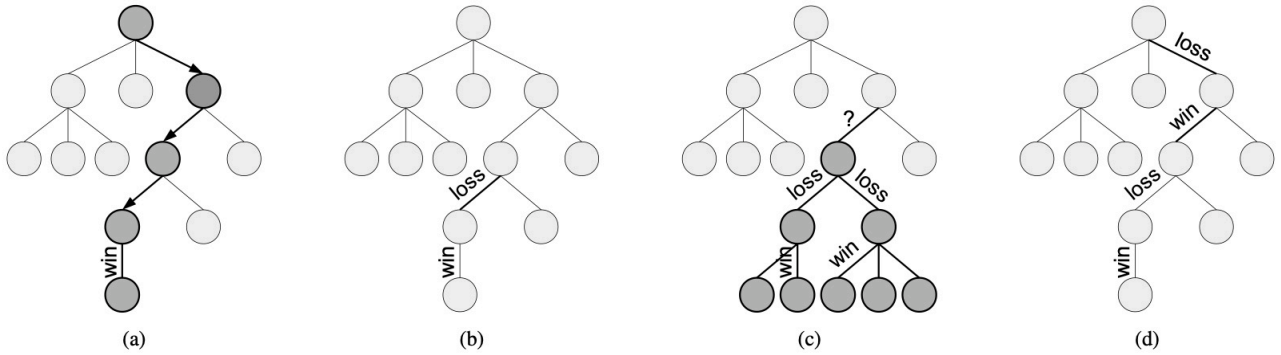


Abbildung 2.19: MiniMax in der *Backpropagation*-Phase
(a) *Selection* und *Expansion* identifizieren einen *proven win*. (b) Die MCTS-Solver *Backpropagation* wird durchgeführt. Ein *proven win* für den Gegner impliziert, einen *proven loss* für den anderen Spieler. (c) Die MCTS-Solver *Backpropagation* wird unterbrochen, da ein Knoten kein *proven loss* aufweist. Anschließend wird eine MiniMax-Suche für diesen Knoten gestartet, die einen *proven loss* für den Knoten erkennt. (d) Die MCTS-Solver *Backpropagation* wird durch das Ergebnis der MiniMax-Suche fortgesetzt. Es wird eine Niederlage für den Wurzelknoten in diesem Teil des Suchbaums bewiesen. (Quelle: Mcts-minimax hybrids [4])

Pseudocode MCTS-MB

Der MCTS-MB Code ist eine Erweiterung von MCTS-Solver. Stößt MCTS-MB während der *Backpropagation* auf eine *loss*, wird eine Alpha-Beta-Suche durchgeführt (siehe Zeile 10).
Findet Alpha-Beta einen Sieg für den Kindknoten, bedeutet dies einen *proven loss* für den Elternknoten und der Beweis kann fortgesetzt werden (siehe Zeilen 1-13).
Im Fall einer Niederlage des Kindknotens handelt es sich um einen *proven win* für den Elternknoten, welcher sofort zurückpropagiert werden kann (siehe Zeilen 14-17). Andernfalls kann nur eine normale Niederlage zurückpropagiert werden (siehe Zeilen 18-21).
MCTS-MB hat einen weiteren Parameter: die Tiefe für die MiniMax-Suche (`Depth`).

```
1  def MCTS_MB(N: MCTS Node, Depth: Integer):
2      Selection und Simulation wie im MCTS-Solver
3
4      if R == ∞:
5          N.score = ∞
6          return N.score
7      else if R == -∞:
8          for child in N.children:
9              if -child.score != R:
10                 result = AlphaBeta(Depth)
11                 if AlphaBeta proves Win for Child:
12                     child.update_score(∞)
13                     continue
14                 else if AlphaBeta proves Loss for Child:
15                     child.update_score(-∞)
16                     N.score = ∞
17                     return N.score
18                 else
19                     R = -1
20                     N.update_score(R)
21                     return R
22         N.score = -∞
23         return N.score
24     else:
25         N.update_score(R)
26         return R
```

Listing 2.6: MCTS-MB Pseudocode (Quelle: Erweiterung von Monte-carlo tree search solver [23])

2.4.4 *sudden deaths* in „Catch The Lion“

Um die Verbesserung durch die Integration der MiniMax-Suche in den Monte-Carlo-Tree-Search-Algorithmus richtig testen zu können, ist es notwendig, dass das gewählte Spiel „Catch The Lion“ eine relevante Dichte an *sudden deaths* innerhalb des kompletten Suchraums aufweist.
Um diese Eigenschaft nachzuweisen, wurde in „MCTS-Minimax Hybrids“ MCTS-Solver 1000-mal gegen sich selbst getestet. Dabei wurde pro Zug auf die Existenz von *sudden-death*-Fallen innerhalb der nächsten bis zu drei Zügen getestet [4].
In Abbildung 2.20 ist zu erkennen, dass im Spiel „Catch The Lion“ zu jeder Phase des Spiels eine hohe Dichte an Fallen vorhanden ist. Sowohl zu Beginn als auch zum Ende des Spiels ist es möglich, das Spiel innerhalb von drei Zügen zu verlieren.

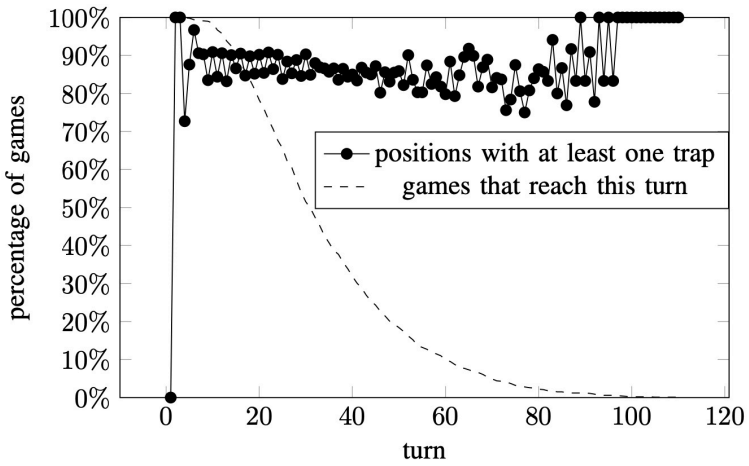


Abbildung 2.20: Dichte der *sudden deaths* in „Catch The Lion“ (Quelle: Mcts-minimax hybrids [4])

Des Weiteren wurde von Hendrik Baier und Mark H. M. Winands auch die Anzahl der unterschiedlichen Arten von *sudden-death*-Fallen für mehrere Spiele getestet. Hier wird nach *level-k traps* unterschieden; also Fallen, die in k Zügen zur Niederlage führen [4].
In den vier getesteten Spielen fällt auf, dass „Catch The Lion“ die größte Anzahl an Fallen in Abbildung 2.21 aufweist.
Ein wichtiger Aspekt einer *sudden-death*-Falle im Kontext von Monte-Carlo-Tree-Search ist, ob die Zufallssimulation diese erkennt oder „in die Falle tappt“. Das bedeutet, dass die Simulation im Falle einer *sudden-death*-Situation das erwartete Ergebnis einer Niederlage zurückgibt oder ein falsches, irreführendes Ergebnis des Sieges liefert. Um dies zu testen, führten Baier und Winands in „MCTS-Minimax Hybrids“ 1000 Zufallssimulationen für jede der erkannten Fallen durch [4].
In Abbildung 2.22 zeigt sich „Catch The Lion“ als das schwierigste Spiel für MCTS mit den meisten *sudden-death*-Fallen, die von der Simulation nicht erkannt wurden.
„Catch The Lion“ besitzt also nicht nur eine hohe Dichte an *sudden-death*-Fallen, sondern diese Fallen führen oft zu falschen Simulationsergebnissen. Somit ist zu erwarten, dass MiniMax-Hybriden mit dem Ziel eines besseren Umgangs mit *sudden-death*-Fallen eine deutlich höhere Spielstärke in „Catch The Lion“ aufweisen sollten als der MCTS-Solver-Algorithmus [4].

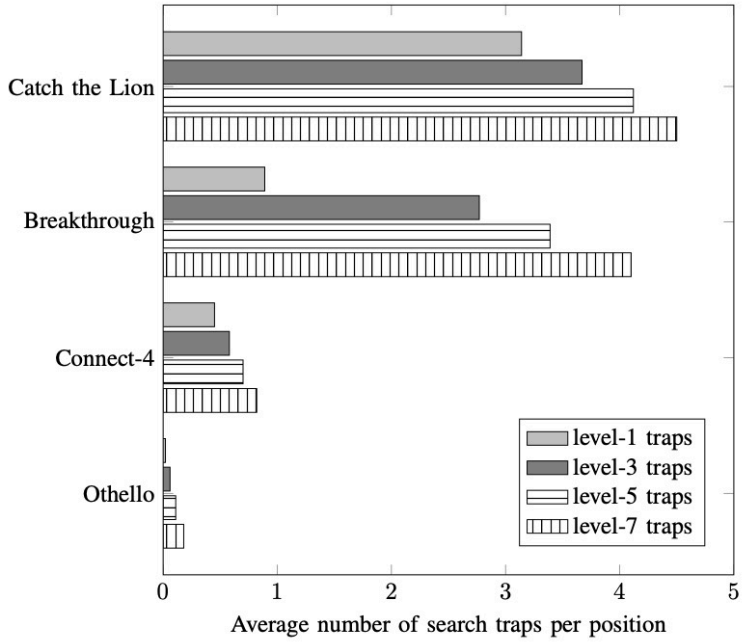


Abbildung 2.21: Durchschnittliche Anzahl von *sudden deaths* (Quelle: Mcts-minimax hybrids [4])

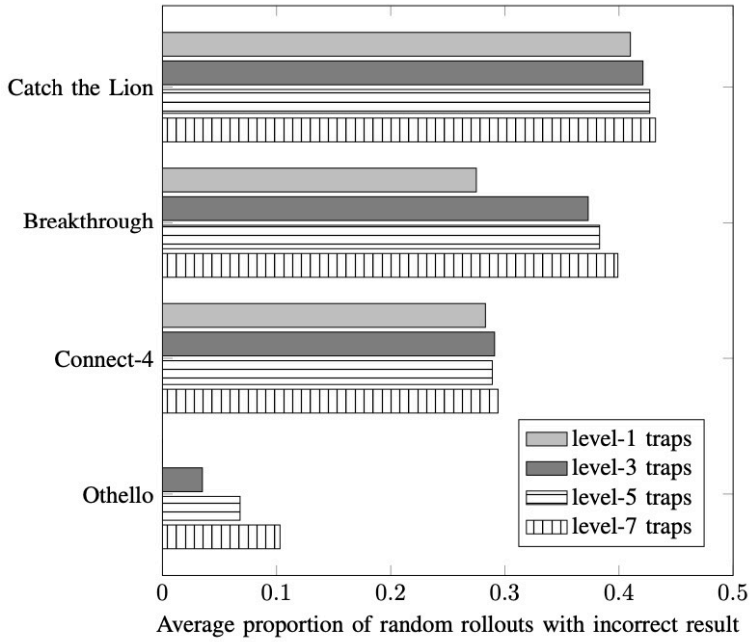


Abbildung 2.22: Durchschnittliche Schwierigkeit von *sudden deaths* (Quelle: Mcts-minimax hybrids [4])

3 Ergebnisse

3.1 Vergleich der MiniMax-Algorithmen

3.1.1 Zeit für Suchtiefe in Startstellung

Da jede Variante des MiniMax-Algorithmus bei einer festen Suchtiefe den gleichen Zug zurückgibt, ist es möglich, MiniMax-Algorithmen anhand der Zeit, die sie für die gleiche Tiefe benötigen, zu vergleichen.
Der Test besteht darin, MiniMax, Alpha-Beta, Alpha-Beta mit Transpositionstabellen und MTD(f) an der Startstellung als neutrale Spielsituation mit fester Tiefe zu vergleichen. Wenn ein Algorithmus deutlich mehr Zeit benötigt als die anderen für eine bestimmte Tiefe, wird er für höhere Tiefen aufgrund des exponentiellen Wachstums von MiniMax-Algorithmen mit der Suchtiefe ausgeschlossen.
In Abbildung 3.1 ist zu erkennen, dass MiniMax ab Tiefe 7 mit mehr als 2 Minuten deutlich länger benötigt als die anderen Varianten.
Ab Tiefe 10 benötigt Alpha-Beta fast 10 Sekunden mehr als Alpha-Beta mit Transpositionstabellen und MTD(f), welche noch unter 12 Sekunden liegen (siehe Abbildung 3.2).
Bei Tiefe 12 steigt die benötigte Zeit für Alpha-Beta mit Transpositionstabellen und MTD(f) deutlich an. Dennoch benötigt MTD(f) nur ungefähr die Hälfte der Zeit im Vergleich zu Alpha-Beta mit Transpositionstabellen.

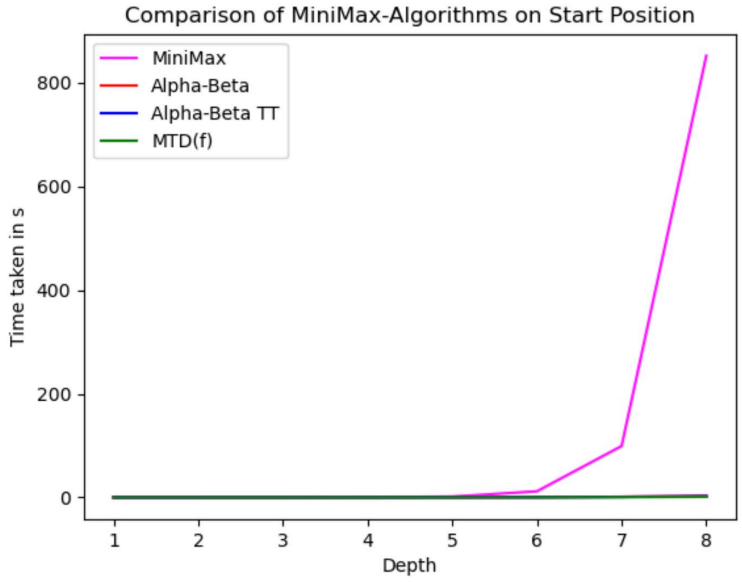


Abbildung 3.1: Startstellung Tiefe 8 (Quelle: eigene Abbildung)

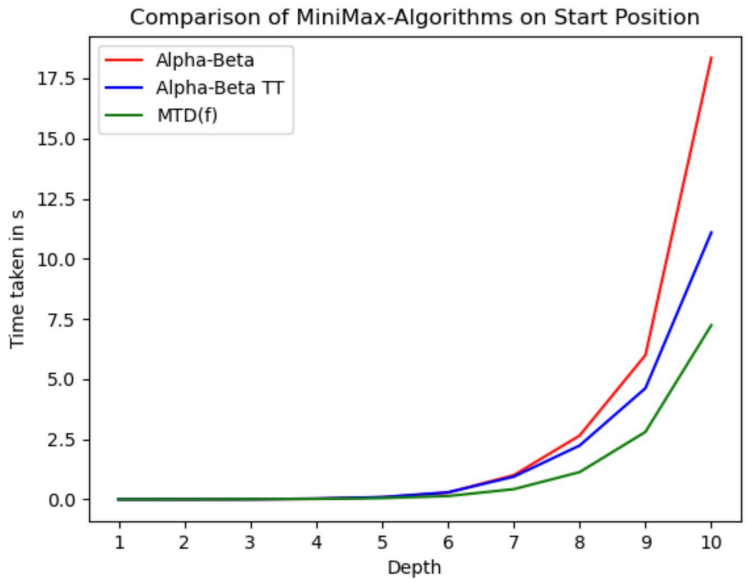


Abbildung 3.2: Startstellung Tiefe 10 (Quelle: eigene Abbildung)

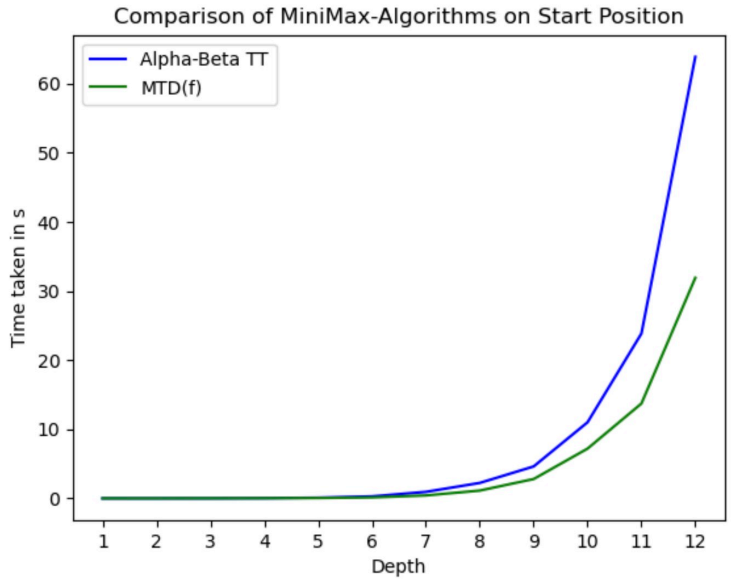


Abbildung 3.3: Startstellung Tiefe 12 (Quelle: eigene Abbildung)

3.1.2 Gegen MCTS-Solver

Um die tatsächliche Spielstärke der MiniMax-Algorithmen zu testen, werden die unterschiedlichen MiniMax-Varianten dem MCTS-Solver-Algorithmus gegenübergestellt. Hierbei wird für die MiniMax-Algorithmen Iterative Deepening eingesetzt. Die MiniMax-Algorithmen bekommen 2 Sekunden pro Zug und der gegenübergestellte MCTS-Solver-Algorithmus bekommt 60 Sekunden pro Zug. Dies hat den Zweck, die Unterschiede zwischen den MiniMax-Algorithmen durch einen stärkeren Gegner zu zeigen.
Pro Gegenüberstellung werden 1000 Spiele simuliert; 500 als weißer Spieler und 500 als schwarzer Spieler.
In Abbildung 3.4 ist zu erkennen, dass von den vier Varianten MiniMax am schlechtesten abschneidet mit einer Siegrate von 46,9% gegen MCTS-Solver.
Im Vergleich dazu macht Alpha-Beta, also MiniMax mit Alpha-Beta-Pruning, einen großen Sprung nach oben zu einer Siegrate von 75%.
Die Einführung von Transpositionstabellen in *Alpha-Beta with TT* führt zu einer weiteren Erhöhung der Siegrate auf 80,4%.
Der Unterschied zwischen Alpha-Beta mit Transpositionstabellen und dem MTD(f)-Algorithmus ist minimal; MTD(f) erreicht eine fast gleiche Siegrate von 80,8%. Im Betracht der 99%-Konfidenzintervalle erscheint die Verbesserung der Spielstärke durch MTD(f) auch minimal; MTD(f) hat das Intervall [79.53%, 82.07%] und Alpha-Beta mit Transpositionstabellen das Intervall [79.11%, 81.69%].

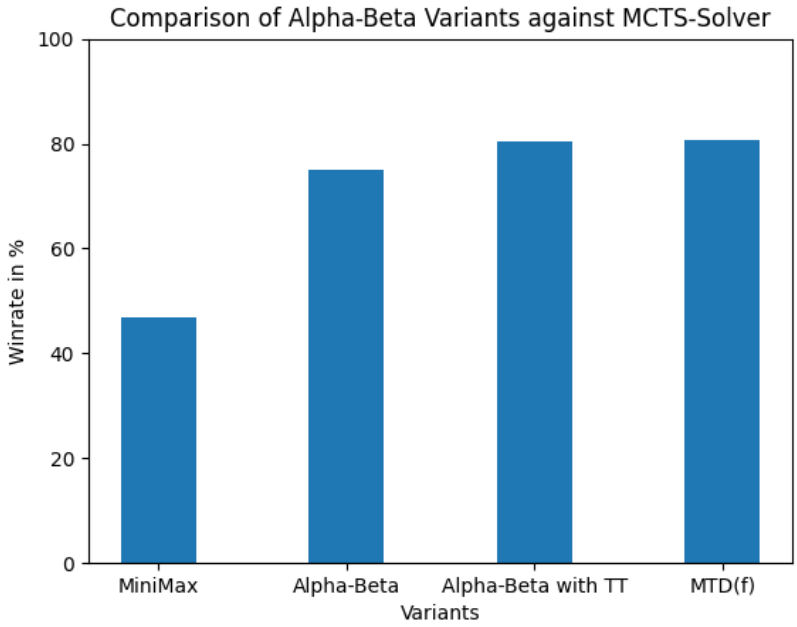


Abbildung 3.4: Vergleich der MiniMax-Algorithmen mit 2 Sekunden pro Zug gegen MCTS-Solver mit 60 Sekunden pro Zug (Quelle: eigene Abbildung)

3.2 Test der MCTS-MiniMax-Hybriden

Um die Spielstärke der MCTS-MiniMax-Hybriden zu testen, werden die Hybrid-Algorithmen gegen den MCTS-Solver sowie die Alpha-Beta-Suche getestet. Für jede Gegenüberstellung werden 1000 Spiele des Spiels „Catch The Lion“ simuliert. Eine Gegenüberstellung besteht zum Beispiel aus MCTS-MR mit einer Suchtiefe von 1 gegen MCTS-Solver. Die Spieler wechseln sich ab, sodass jeder Algorithmus am Ende 500 Spiele als weißer Spieler und 500 Spiele als schwarzer Spieler gespielt hat.
In der Gegenüberstellung mit MCTS-Solver bekommen sowohl der MCTS-MiniMax-Hybrid als auch der MCTS-Solver 2 Sekunden pro Zug. So können die Hybrid-Algorithmen untereinander verglichen werden, sowie auf ein erhöhte Spielstärke gegen den MCTS-Solver getestet werden.
Um den Einfluss von Zeit auf die Spielstärke zu testen, werden zusätzlich die gleichen Gegenüberstellungen auch mit 20 Sekunden pro Zug getestet.
Die MCTS-MiniMax-Hybrid-Algorithmen werden außerdem noch gegen Alpha-Beta mit der festen Suchtiefe von 4 getestet. In dieser Gegenüberstellung bekommen die Hybrid-Algorithmen 20 Sekunden pro Zug, Alpha-Beta hat aufgrund der festen Tiefe kein Zeitlimit. Sinn der Tests ist es, einen weitern Vergleichspunkt mit einem Algorithmus zu erhalten, welcher nicht der Monte-Carlo-Tree-Search-Logik folgt.

3.2.1 MCTS-MR

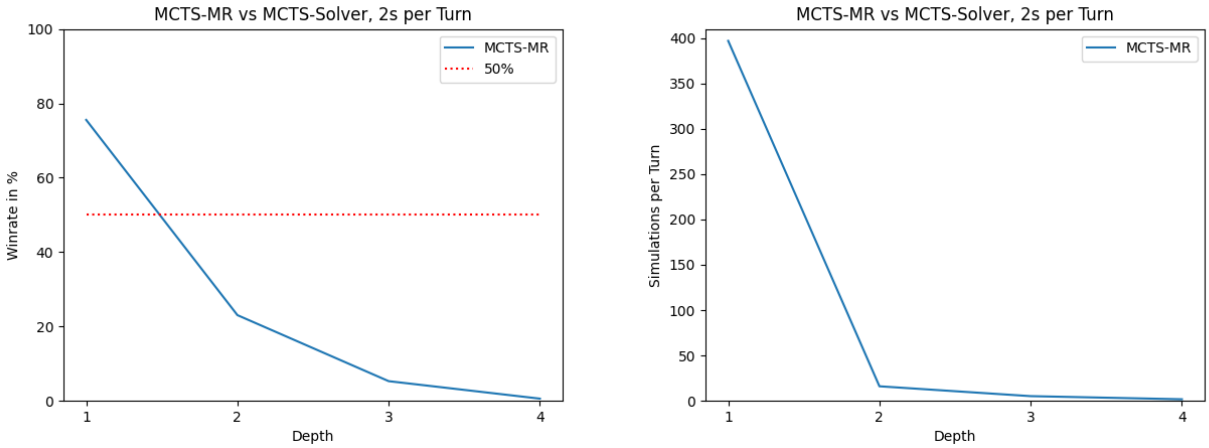
Der MCTS-MR-Algorithmus hat als Parameter die Tiefe der Alpha-Beta-Suche, die während der Simulationsphase häufig aufgerufen wird. Im Rahmen dieser Arbeit werden die Tiefen 1 bis 4 untersucht.

MCTS-MR gegen MCTS-Solver (2 Sekunden pro Zug)

In Abbildung 3.5a ist zu erkennen, dass MCTS-MR mit einer Suchtiefe von 1 deutlich gegen den MCTS-Solver gewinnt. Die 75.5%-ige Siegesrate von MCTS-MR-1 liegt in einem 99%-Konfidenzintervall von [73.99%, 77.01%]. Dagegen liegen MCTS-MR mit den Suchtiefen 2 bis 4 deutlich unter der 50%-Marke und verlieren einen Großteil ihrer Spiele gegen den MCTS-Solver.
In den Spielen gegen den MCTS-MR simuliert der MCTS-Solver durchschnittlich 718 Spiele pro Zug; beinahe das Doppelte der 397 simulierten Spiele von MCTS-MR-1 und deutlich mehr als die 16 bis 2 simulierten Spiele von MCTS-MR-2, MCTS-MR-3 und MCTS-MR-4 (siehe Abbildung 3.5b).

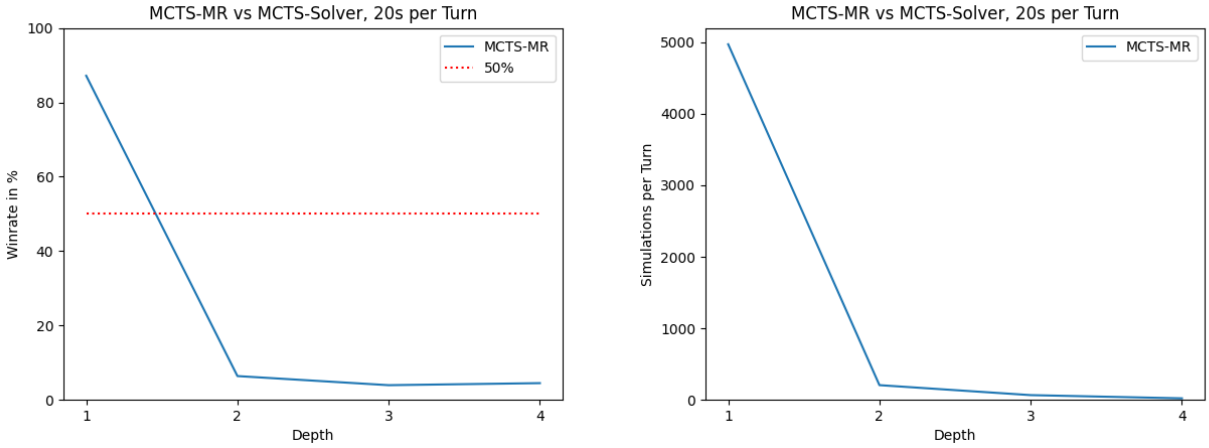
MCTS-MR gegen MCTS-Solver (20 Sekunden pro Zug)

In Abbildung 3.6a ist zu erkennen, dass durch die erhöhte Zeit pro Zug sich die Tendenz aus Abbildung 3.5a weiter verstärken. Mit 20 Sekunden pro Zug ist MCTS-MR-1 in der Lage 87,1% der Spiele gegen MCTS-Solver zu gewinnen, 11,6% mehr als mit nur 2 Sekunden pro Zug.
Die Siegrate von MCTS-MR mit Suchtiefe 2 sinkt jedoch um 16,6% auf ein Tief von 6,4%. Zudem fällt auf, dass die Siegrate von Suchtiefe 4 leicht ansteigt.
Der Verlauf der Anzahl von Simulation ist für 20 Sekunden pro Zug ähnlich zu dem für 2 Sekunden pro Zug (siehe Abbildung 3.5b und 3.6b)



(a) Siegrate (b) Simulationsanzahl pro Zug

Abbildung 3.5: MCTS-MR gegen MCTS-Solver, 2s pro Zug (Quelle: eigene Abbildung)

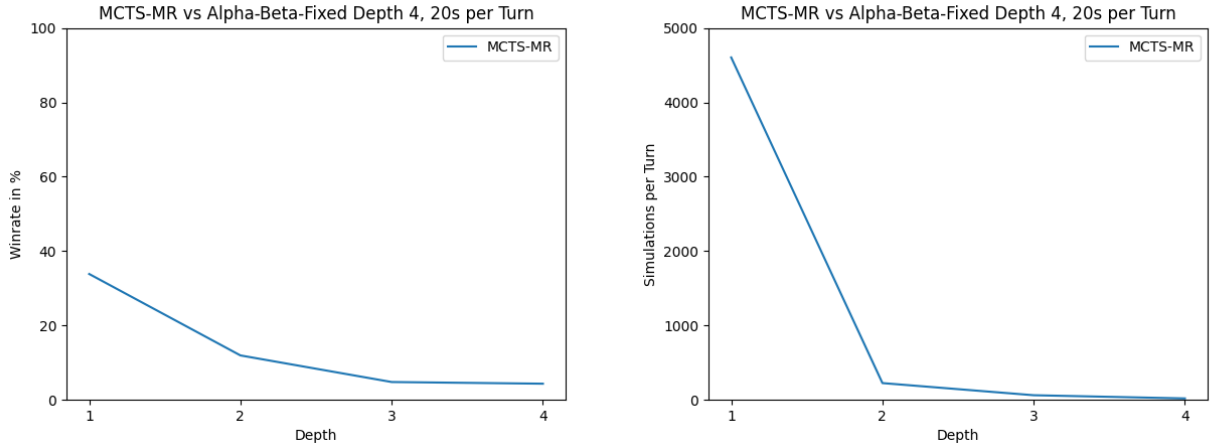


(a) Siegrate (b) Simulationsanzahl pro Zug

Abbildung 3.6: MCTS-MR gegen MCTS-Solver, 20s pro Zug (Quelle: eigene Abbildung)

MCTS-MR gegen Alpha-Beta Tiefe 4 (20 Sekunden pro Zug)

Monte-Carlo-Tree-Search mit MiniMax-Rollout ist gegen Alpha-Beta weitaus weniger erfolgreich als gegen MCTS-Solver. In Abbildung 3.7a ist zu erkennenn, dass keine Variante eine Siegrate von mehr als 50% erreicht.
MCTS-MR mit Suchtiefe 1 erreicht erneut die höchste Siegrate, hier 33,8%, gefolgt von Suchtiefe 2 mit 11,9%. Der Verlauf der Siegrate ist ähnlich dem Verlauf gegen MCTS-Solver. Die Siegrate sinkt mit steigender Tiefe.



(a) Siegrate

(b) Simulationsanzahl pro Zug

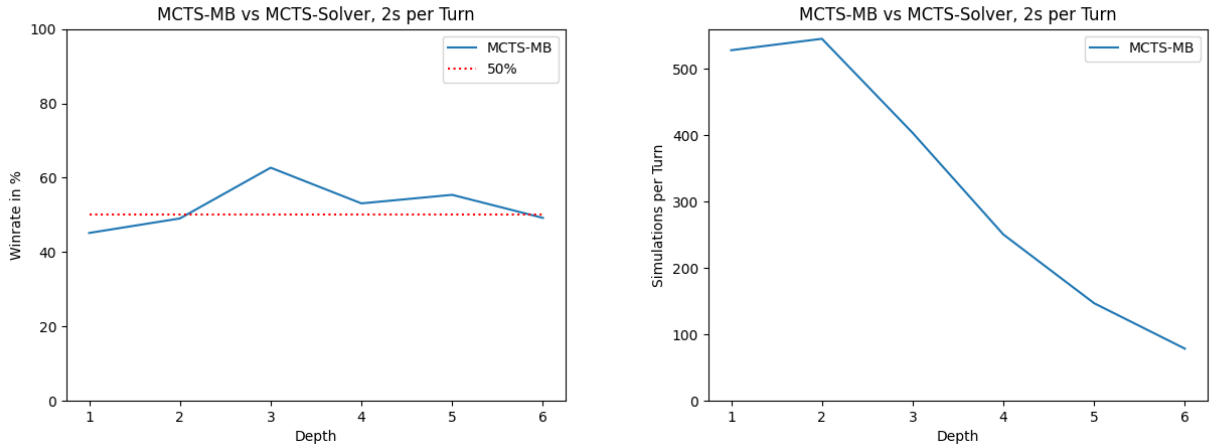
Abbildung 3.7: MCTS-MR gegen Alpha-Beta Tiefe 4, 20s pro Zug (Quelle: eigene Abbildung)

3.2.2 MCTS-MB

Der MCTS-MB-Algorithmus hat als Parameter die Tiefe des Alpha-Beta-Algorithmus, der in der Backpropagation-Phase aufgerufen wird. Im Rahmen dieser Arbeit werden Tiefen von 1 bis 6 untersucht.

MCTS-MB gegen MCTS-Solver (2 Sekunden pro Zug)

Unter den MCTS-MB-Algorithmen sind die Tiefen 3 bis 5 in der Lage, die 50%-Hürde zu überwinden und häufiger gegen den MCTS-Solver zu gewinnen als zu verlieren (siehe Abbildung 3.8a). Besonders hervorzuheben ist MCTS-MB-3 als der erfolgreichste der MCTS-MB-Algorithmen mit einer Gewinnrate von 62,7% und einem 99%-Konfidenzintervall von [60,79%, 64,61%].
Die durchschnittliche Anzahl der Simulationen von MCTS-MB mit Suchtiefen 1 und 2 liegt knapp unter den durchschnittlichen 594 Simulationen des MCTS-Solvers. In Abbildung 3.5b ist zu erkennen, dass ab Suchtiefe 3 mit steigender Suchtiefe die durchschnittliche Anzahl der Simulationen abnimmt, bis hin zu durchschnittlich 79 Spielsimulationen pro Zug bei MCTS-MB-6.



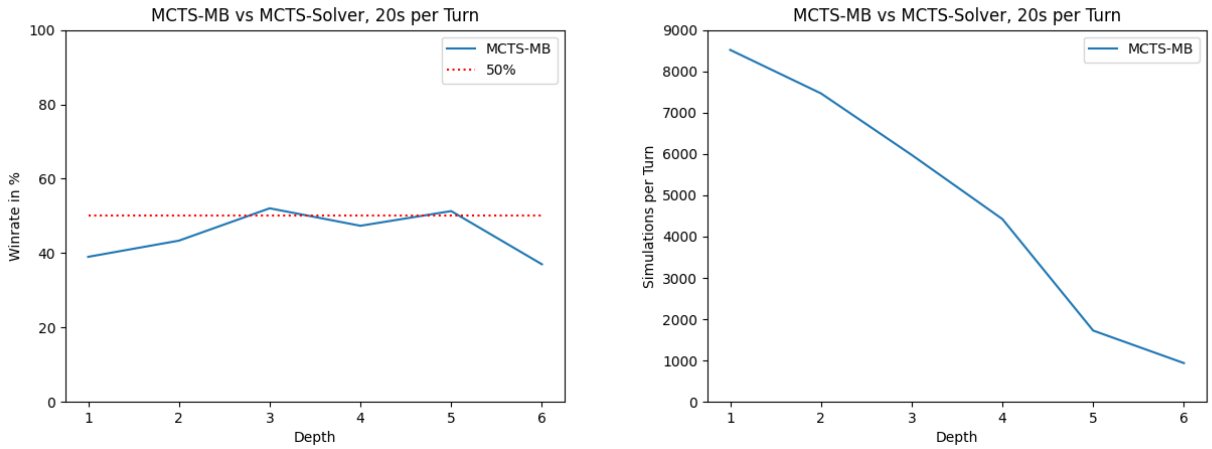
(a) Siegrate

(b) Simulationsanzahl pro Zug

Abbildung 3.8: MCTS-MB gegen MCTS-Solver, 2s pro Zug (Quelle: eigene Abbildung)

MCTS-MB gegen MCTS-Solver (20 Sekunden pro Zug)

Die Erhöhung der Zeit pro Zug hat den Effekt, dass der Unterschied in Spielstärke zwischen MCTS-MB und MCTS-Solver sinkt. Von MCTS-MB sind nur noch die Tiefen 3 und 5 in der Lage mehr als 50 % der Spiele zu gewinnen. Die Siegrate von 52% für MCTS-MB-3 und 51,3% für MCTS-MB-5 ist geringer als die Siegrate von 62,7% und 55,4% die MCTS-MB-3 und MCTS-MB-5 in der Gegenüberstellung mit 2 Sekunden pro Zug erreichen konnte (siehe Abbildung 3.8 und 3.9)
Die durchschnittliche Anzahl der Simulationen nimmt auch bei 20 Sekunden pro Zug mit der Suchtiefe ab.

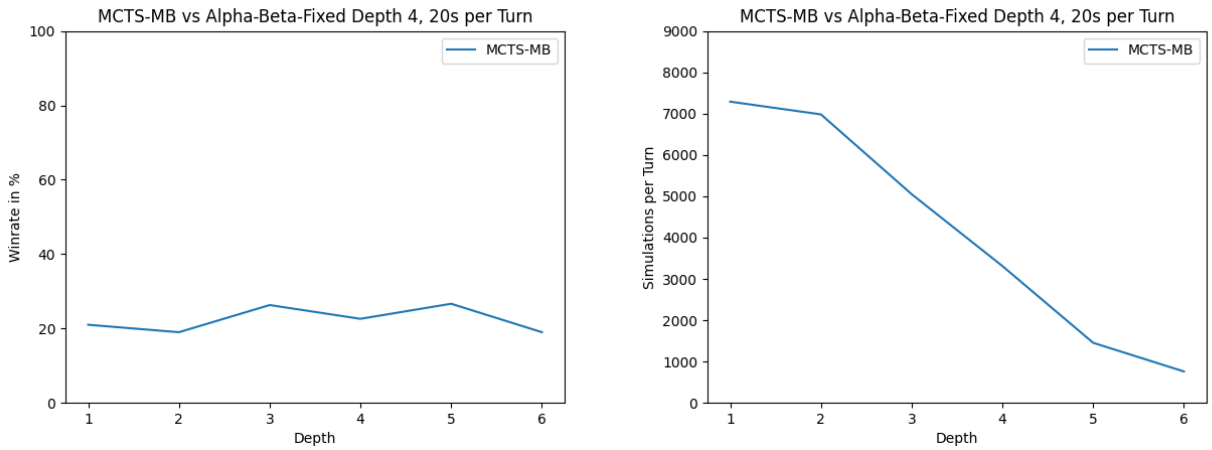


(a) Siegrate (b) Simulationsanzahl pro Zug

Abbildung 3.9: MCTS-MB gegen MCTS-Solver, 20s pro Zug (Quelle: eigene Abbildung)

MCTS-MB gegen Alpha-Beta Tiefe 4 (20 Sekunden pro Zug)

Die Siegrate aller Varianten des MCTS-MB Algorithmus gegen Alpha-Beta ist weit unter 50%. Dennoch haben wie in der Gegenüberstellung mit MCTS-Solver wieder die Tiefen 3 und 5 die höchste Siegrate, mit 26,3% und 26,6%, Tiefe 5 eine leicht höhere Siegrate als Tiefe 3.



(a) Siegrate (b) Simulationsanzahl pro Zug

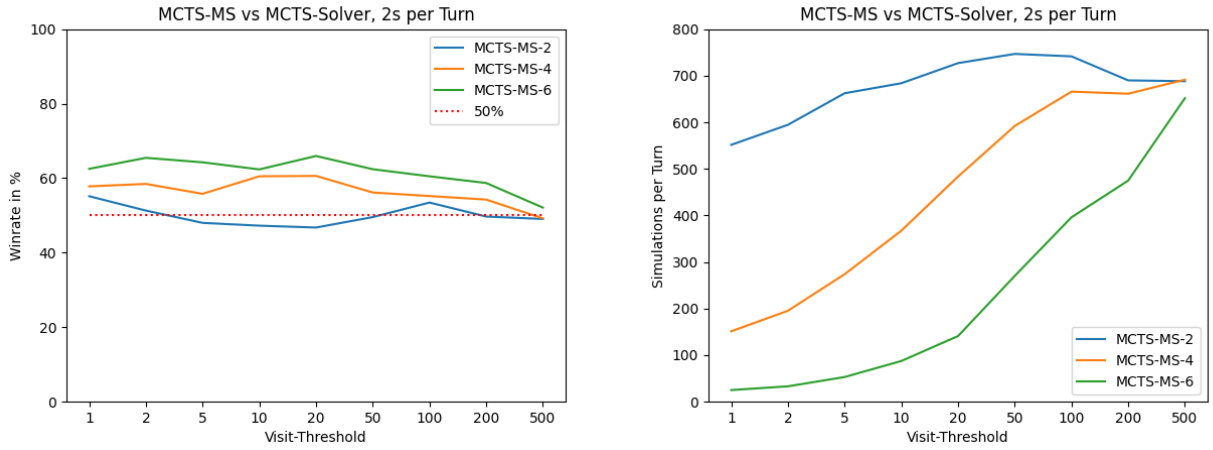
Abbildung 3.10: MCTS-MB gegen Alpha-Beta Tiefe 4, 20s pro Zug (Quelle: eigene Abbildung)

3.2.3 MCTS-MS

Der MCTS-MS-Algorithmus hat als Parameter die Tiefe des Alpha-Beta-Algorithmus, der in der *Selection*-Phase aufgerufen wird. Im Rahmen dieser Arbeit werden die Tiefen 2, 4 und 6 untersucht. Zusätzlich hat der MCTS-MS-Algorithmus einen zweiten Parameter, das *Visit-Threshold*, welches bestimmt, ab wie vielen Besuchen die Alpha-Beta-Suche während der *Selection*-Phase aufgerufen wird. Es werden die *Visit-Thresholds* 1, 2, 5, 10, 20, 50, 100, 200 und 500 untersucht.

MCTS-MS gegen MCTS-Solver (2 Sekunden pro Zug)

Beim Vergleich der drei Tiefen in Abbildung 3.5a fällt auf, dass MCTS-MS-6 am besten abschneidet, während MCTS-MS-2 am schlechtesten abschneidet.
Von den MCTS-MS-2-Varianten schneidet MCTS-MS-2-1 am besten ab mit einer Gewinnrate von 55,1% und einem 99%-Konfidenzintervall von [53,08%, 57,12%].
Unter den MCTS-MS-4-Varianten schneidet MCTS-MS-4-20 am besten ab mit einer Gewinnrate von 60,6% und einem 99%-Konfidenzintervall von [58,65%, 62,55%].
Allerdings schneidet unter allen MCTS-MS-Algorithmen MCTS-MS-6-20 am besten ab mit einer Gewinnrate von 66% und einem 99%-Konfidenzintervall von [64,17%, 67,83%].
In den Spielen gegen die MCTS-MS-Varianten hat MCTS-Solver durchschnittlich 749 Spiele simuliert. In Abbildung 3.11b ist zu erkennen, dass mit steigendem *Visit-Threshold* auch die Anzahl der simulierten Spiele von MCTS-MS steigt. MCTS-MS-2 simuliert die meisten Spiele, und mit einem *Visit-Threshold* von 50 oder 100 sogar ähnlich viele wie MCTS-Solver. Außerdem fällt auf, dass durchweg MCTS-MS-6 die wenigsten Spiele simuliert.



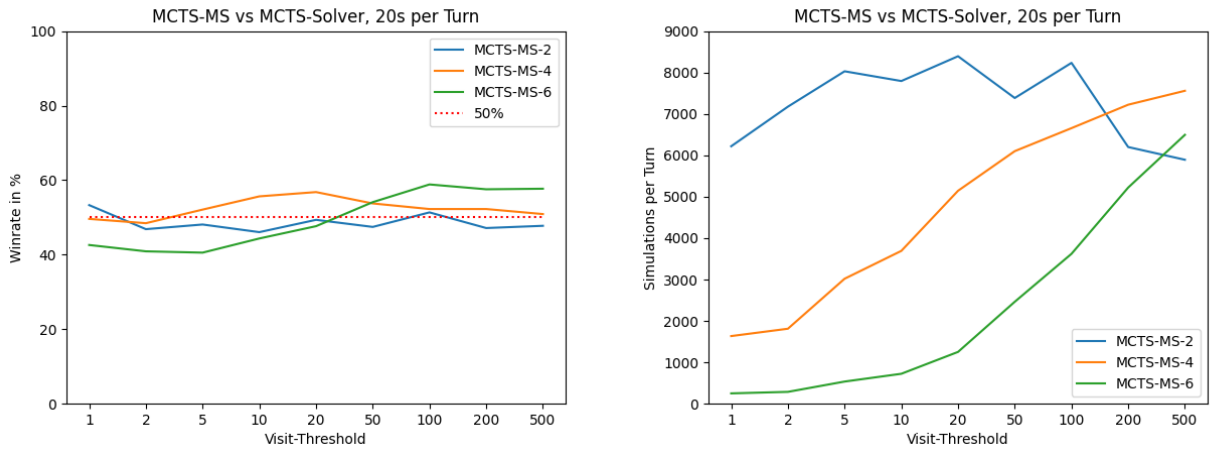
(a) Siegrate

(b) Simulationsanzahl pro Zug

Abbildung 3.11: MCTS-MS gegen MCTS-Solver, 2s pro Zug (Quelle: eigene Abbildung)

MCTS-MS gegen MCTS-Solver (20 Sekunden pro Zug)

Beim Vergleich der Abbildungen 3.11a und 3.12a fällt auf, dass die Zeit pro Zug einen starken Einfluss auf die Spielstärke der verschiedenen MCTS-MS-Varianten hat.
MCTS-MS mit einer Suchtiefe von 6 ist nicht mehr die Variante mit der höchsten Gewinnrate für jedes *Visit-Threshold*. Für die *Visit-Thresholds* von 1 bis 20 liegt die Gewinnrate von MCTS-MS-6 unter 50%. Ab dem *Threshold* 50 hat MCTS-MS-6 eine höhere Gewinnrate als 50%, wobei die höchste Gewinnrate bei einem *Visit-Threshold* von 100 liegt, nämlich bei 58,8% mit einem 99%-Konfidenzintervall von [56,82%, 60,78%].
Der Verlauf von MCTS-MS-2 ähnelt stark dem mit 2 Sekunden pro Zug, wobei die Gewinnraten für die *Visit-Thresholds* 1 und 100 über 50% liegen, aber der Rest unter 50%.
Die Gewinnrate von MCTS-MS-4 in Abbildung 3.12a steigt erst mit dem *Visit-Threshold* auf einen Höchstwert von 56,8% für MCTS-MS-4-20, sinkt dann jedoch wieder.
Der Verlauf der Anzahl der Simulationen in Abbildung 3.12b ähnelt sehr dem Verlauf in Abbildung 3.11b.



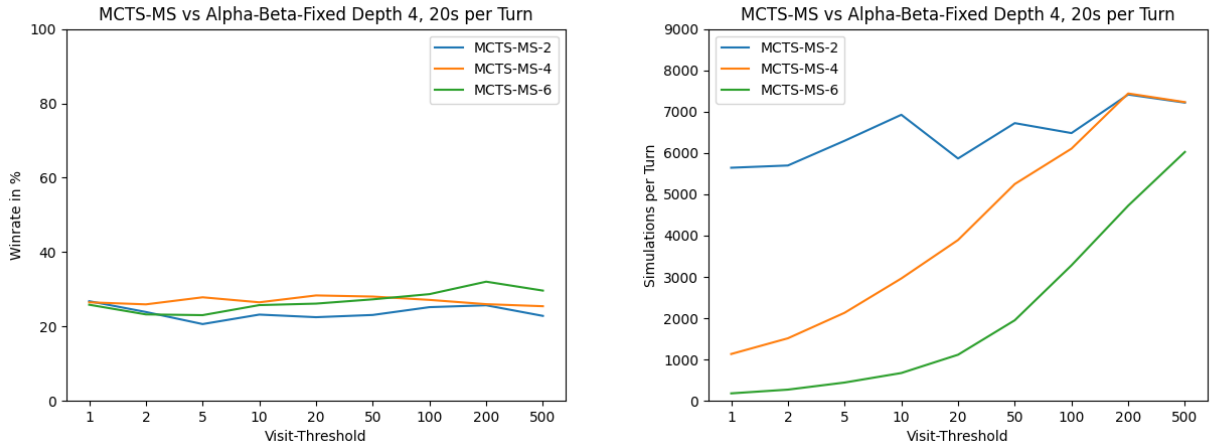
(a) Siegrate

(b) Simulationsanzahl pro Zug

Abbildung 3.12: MCTS-MS gegen MCTS-Solver, 20s pro Zug (Quelle: eigene Abbildung)

MCTS-MS gegen Alpha-Beta Tiefe 4 (20 Sekunden pro Zug)

Im Spiel gegen Alpha-Beta liegt die Siegrate aller MCTS-MS Varianten zwischen 20 und 30%, dabei ist der Verlauf der Siegrate dem Verlauf aus Abbildung 3.12a ähnlich.
Die Siegrate von MCTS-MS-6 steigt mit der Erhöhung des *Visit-Threshold* an, wie in der Gegenüberstellung mit MCTS-Solver mit 20 Sekunden pro Zug.
Auch MCTS-MS-2 hat wieder einen U-förmigen Verlauf mit den Höhepunkten MCTS-MS-2-1 und MCTS-MS-2-200.
Der Verlauf von MCTS-MS-4 auch wenn sehr abgeschächte, ist dem Verlauf aus Abbildung 3.12a ähnlich. Die Siegrate steigt erst mit dem *Threshold*, sinkt aber ab dem Höhepunkt MCTS-MS-4-20 wieder.



(a) Siegrate

(b) Simulationsanzahl pro Zug

Abbildung 3.13: MCTS-MS gegen Alpha-Beta Tiefe 4, 20s pro Zug (Quelle: eigene Abbildung)

3.3 MCTS-MiniMax-Hybriden gegen MiniMax

Zur Bestimmung des insgesamt spielstärksten Algorithmus im Spiel „Catch The Lion“ werden die Algorithmen mit der höchsten Siegrate aus den Abschnitten Vergleich der MiniMax-Algorithmen und Test der MCTS-MiniMax-Hybriden gegeneinander getestet.
Unter den MiniMax-Algorithmen hat MTD(f) die höchste Siegrate. Aus den MCTS-MiniMax-Hybriden wird jeweils die Variante mit der höchsten Siegrate gewählt. Zum Beispiel hat Monte-Carlo-Tree-Search mit MiniMax in der Simulation-Phase die Suchtiefe 1 als die Variante mit der höchsten Siegrate.
Es spielen also die besten MCTS-MiniMax-Hybriden gegen MTD(f). Beiden Algorithmen wird die gleiche Zeit pro Zug gewährt, jeweils 2 und 20 Sekunden. Es werden 1000 Partien simuliert:

Tabelle 3.1: Siegrate gegen MTD(f) (Quelle: eigene Tabelle)

Algorithmus	2s pro Zug	20s pro Zug
MCTS-Solver	2%	2,2%
MCTS-MR-1	2,4%	5,2%
MCTS-MB-3	2,9%	2,9%
MCTS-MS-2-1	1,9%	4,5%
MCTS-MS-4-20	2,9%	5,1%
MCTS-MS-6-20	2%	/
MCTS-MS-6-200	/	6,6%

In der Tabelle 3.3 ist zu erkennen, dass keiner der Hybrid-Algorithmen eine Siegrate von 10% erreicht. Somit gewinnt MTD(f) in jeder Gegenüberstellung mehr als 90% der Partien. Zudem fällt auf, dass einige der Hybriden mit zunehmender Bedenkzeit pro Zug an Spielstärke gewinnen, besonders MCTS-MS-6 verbessert sich um 4,6%. Im Gegensatz dazu verbessert sich der Basisalgorithmus MCTS-Solver nur um 0,2%.

3.4 Technische Details

Die Programmierung der Experimente, also die Implementierung des Spiels sowie der Algorithmen, wurde in Python geschrieben.
Die Experimente wurde auf dem Hydra Cluster der TU Berlin durchgeführt (siehe Anhang 6.0.1). Es wurden die *partitions* `cpu-2h` und `cpu-2d` genutzt.

4 Diskussion

4.1 Vergleich der MiniMax-Algorithmen

4.1.1 Zeit für Suchtiefe in Startstellung

In dem Abschnitt „Zeit für Suchtiefe in Startstellung“ hat sich wie erwartet gezeigt, dass die Verbesserungen des MiniMax-Algorithmus einen deutlichen Einfluss auf die benötigte Zeit für höhere Tiefen haben.
In Grafik 3.1 ist deutlich zu erkennen, dass der grundlegende Basis-MiniMax-Algorithmus ab einer Tiefe von 7 mit Abstand am längsten braucht. Da es sich um MiniMax ohne Verbesserungen handelt, ist dieses Ergebnis ebenfalls wie erwartet.
Aus Abbildung 3.1 lässt sich auch erkennen, dass durch die in Alpha-Beta erzeugten *cutoffs* ein erheblicher Teil des Suchbaums nicht erkundet werden muss und so stark Zeit eingespart werden kann. Dies führt dazu, dass die Alpha-Beta-Suche bereits ab Tiefe 7 drastisch schneller als der Basis-MiniMax-Algorithmus ist.
In Abbildung 3.2 zeigt sich, dass ab einer Suchtiefe von 9 die Nutzung von Transpositionstabellen einen merkbaren Effekt auf die benötigte Zeit hat. Die eingesparte Rechenzeit durch das Auslesen von bereits erkundeten Spielpositionen aus dem Speicher im Gegensatz zur kompletten Neuberechnung führt dazu, dass die normale Alpha-Beta-Suche für Tiefe 10 mehr als 17 Sekunden braucht, während Alpha-Beta mit Transpositionstabellen weniger als 12 Sekunden benötigt.
Abbildung 3.3 belegt, dass die fokussierte Suche von MTD(f) mittels Nullfenster-Suche ab einer Tiefe von 9 zu einer schnelleren Berechnung des optimalen MiniMax-Wertes führt als die Alpha-Beta-Suche mit Transpositionstabellen.
Unter der Annahme, dass die MiniMax-Suche mit höherer Suchtiefe definitiv besser ist als die MiniMax-Suche mit niedriger Suchtiefe (siehe Anhang 6.3), lässt sich die These aufstellen, dass MTD(f) die beste Variante der MiniMax-Algorithmen ist, da MTD(f) die untersuchten Tiefen am schnellsten berechnen konnte.

4.1.2 Gegen MCTS-Solver

In der Gegenüberstellung mit dem MCTS-Solver konnte getestet werden, inwieweit sich die Geschwindigkeit beim Berechnen der verschiedenen Tiefen auf die tatsächliche Spielstärke der MiniMax-Varianten auswirkt. Bei diesem Vergleich wurde Iterative Deepening verwendet. Ist ein MiniMax-Algorithmus schneller im Berechnen höhere Tiefen, wie im Abschnitt „Zeit für Suchtiefe in Startstellung“ gezeigt, so sollte er bei gleicher Zeit im Iterative Deepening eine höhere Tiefe erreichen und damit auch eine höhere Spielstärke aufweisen.
In Abbildung 3.4 zeigt sich deutlich, dass der langsamste der Algorithmen, die MiniMax-Suche, auch am schlechtesten abschneidet mit nicht einmal 50% Siegrate gegen den MCTS-Solver.
Die erhöhte Geschwindigkeit von Alpha-Beta im Berechnen der verschiedenen Suchtiefen führt dazu, dass der Algorithmus durch Iterative Deepening höhere Tiefen erreichen kann und somit einen deutlichen Anstieg in der Spielstärke auf eine 75% Siegrate verzeichnet.
Wie zu erwarten, führt die Einführung von Transpositionstabellen zu einer weiteren Steigerung der Spielstärke, auch wenn der Anstieg mit einer Erhöhung auf 80% relativ moderat ausfällt.
Das Verhalten von MiniMax, Alpha-Beta und Alpha-Beta mit Transpositionstabellen ist wie zu erwarten; Zeiteinsparungen während der Berechnung des optimalen MiniMax-Wertes führen zu einer erhöhten Spielstärke im Iterative Deepening. MTD(f) unterbricht jedoch dieses Verhalten. Der Unterschied von 0,4% zu Alpha-Beta mit Transpositionstabellen ist im Prinzip nicht existent, sodass beide Algorithmen die gleiche Spielstärke aufweisen. In Abbildung 3.3 ist jedoch zu erkennen, dass MTD(f) effizienter im Berechnen der höheren Tiefen ist als Alpha-Beta mit Transpositionstabellen. Hier stellt sich die Frage, warum sich dieses Verhalten nicht auf eine erhöhte Spielstärke für MTD(f) überträgt, wie es für die restlichen Verbesserungen von MiniMax gezeigt wurde.
In der Gegenüberstellung mit dem MCTS-Solver wurden die MiniMax-Algorithmen im Iterative Deepening mit 2 Sekunden getestet. Jedoch zeigt sich in Abbildung 3.3, dass der Unterschied zwischen Alpha-Beta mit Transpositionstabellen und MTD(f) erst ab einer Tiefe von 9 merkbar und ab Tiefe 11 signifikant wird. Für eine Tiefe von 9 benötigt MTD(f) bereits mehr als 2 Sekunden, selbst in der Startstellung, die im Vergleich zu vielen anderen Stellungen im Spiel einen moderaten *branching factor* aufweist. Somit erreichen sowohl MTD(f) als auch Alpha-Beta mit Transpositionstabellen im normalen Spiel konsistent eine Tiefe von 8 im Iterative Deepening, sind jedoch beide nicht in der Lage, die Berechnung von Tiefe 9 im Rahmen von 2 Sekunden pro Zug abzuschließen. Um den Unterschied zwischen MTD(f) und Alpha-Beta mit Transpositionstabellen genauer festzustellen, wäre es daher notwendig, die beiden Algorithmen mit mehr Zeit pro Zug zu testen. Hierfür ist jedoch auch ein stärkerer Gegner als der MCTS-Solver erforderlich, damit die Unterschiede deutlich erkennbar sind. Diese Analyse übersteigt jedoch den Rahmen dieser Arbeit und wird daher nicht durchgeführt.

4.2 Test der MCTS-MiniMax-Hybriden

4.2.1 MCTS-MR

Der Test von MCTS-MR hat gezeigt, dass eine informierte Simulation durch Alpha-Beta die Spielstärke des Monte-Carlo-Tree-Search steigern kann, wie am Erfolg von MCTS-MR-1 zu erkennen ist. Dennoch hat sich auch gezeigt, dass eine informiertere Simulation mit erhöhtem Zeitaufwand einhergeht. Schon MCTS-MR-1 schafft nur ungefähr die Hälfte der durchschnittlichen 718 Spiele des MCTS-Solvers, doch reicht hier die erhöhte Qualität der Simulationen aus, um die reduzierte Quantität mehr als auszugleichen. Das ändert sich jedoch für die restlichen Tiefen. Hier steigt der Zeitaufwand für die Simulationen stark an aufgrund des exponentiell wachsenden Zeitaufwands der Alpha-Beta-Suche mit steigender Tiefe. Die Konsequenz ist, dass die Anzahl der Simulationen entsprechend stark sinkt. Mit einer so geringen Anzahl von Simulationen ist der Monte-Carlo-Tree-Search nicht in der Lage, eine ausreichende Statistik aufzubauen, um den optimalen Zug zu bestimmen. Es ist also durchaus logisch, dass MCTS-MR mit Suchtiefen 2 bis 4 eine Siegrate weit unter 50% haben. Bemerkenswert ist jedoch, dass obwohl MCTS-MR-2 nur durchschnittlich 16 Spiele pro Zug simuliert, die Qualität dieser Simulationen ausreicht, um ungefähr ein Viertel der Spiele gegen den MCTS-Solver zu gewinnen.
Mit einer erhöhten Zeit von 20 Sekunden pro Zug bestätigen sich diese Erkenntnisse. MCTS-MR-1 ist in der Lage, mit mehr Zeit durch die informierte Simulation eine noch bessere Statistik aufzubauen, sodass die Siegrate gegen den MCTS-Solver weiter steigt. Die Probleme von MCTS-MR mit höheren Suchtiefen potenzieren sich jedoch auch. MCTS-MR-2 erreicht mit 20 Sekunden pro Zug zwar eine bessere Anzahl von Simulationen mit 205 pro Zug, im Spiel gegen den MCTS-Solver mit 8000 Simulationen reicht der Vorteil der stark informierten Suche nicht aus, um den erheblichen Nachteil an Simulationsmangel auszugleichen, so auch MCTS-MR-3 und MCTS-MR-4.
Auch in der Gegenüberstellung mit Alpha-Beta mit fester Tiefe ist der Unterschied zwischen den Varianten zwar kleiner als gegen den MCTS-Solver, aber dennoch zeichnet sich hier MCTS-MR-1 wieder als die beste Variante ab.
Im Fall von MCTS-MR mit Suchtiefe 1 gibt es noch einen Punkt, der beachtet werden sollte. Es muss beachtet werden, wie viel des Erfolgs auf die Alpha-Beta-Funktionalität zurückzuführen ist. Alpha-Beta mit Suchtiefe 1 bewertet nicht mehr als jeden möglichen Zug mit einer Evaluation; im Gegensatz zu Alpha-Beta mit höherer Suchtiefe, bei der auch die Folgezüge betrachtet werden. Es ist bekannt, dass der Monte-Carlo-Tree-Search durch eine heuristikgetriebene Simulationsstrategie verbessert werden kann [10]. Somit stellt sich die Frage, ob der Erfolg von MCTS-MR-1 auf die Alpha-Beta-Funktionalität zurückzuführen ist oder ob es sich hier nur um eine heuristikgetriebene Simulationsstrategie handelt, mit welcher MCTS-MR mit Suchtiefe 1 Erfolg aufweisen konnte.
Weiterführend wäre also der Vergleich zwischen MCTS-MR und einem Monte-Carlo-Tree-Search mit heuristischer Simulationsstrategie sinnvoll, dies wird jedoch nicht im Rahmen dieser Arbeit behandelt.
Die von Monte-Carlo-Tree-Search mit MiniMax *Simulation* verwendete Alpha-Beta-Suche benutzt als Evaluationsfunktion, eine simple Sieg/Niederlage Evaluation. Interssante wäre also, inwiefern sich der Algorithmus entwickelt, wenn man eine komplexere Evaluationsfunktion verwendet. In dieser Arbeit wurde einige Simulation dazu durchgeführt; aufgrund von unerklärlichen Verhalten über die Zeit pro Zug (siehe Anhang 6.1 und 6.2) , wird das aus zeitlichen Gründen nicht weiter thematisert. Der Einfluss einer komplexere Evaluationsfunktion sollte dennoch zukünftig untersucht werden.

4.2.2 MCTS-MB

In der Gegenüberstellung von MCTS-MB mit dem MCTS-Solver hat sich gezeigt, dass die Verwendung der Alpha-Beta-Suche während der *Backpropagation* durchaus eine Verbesserung darstellt. Insbesondere zeigt sich, dass MCTS-MB-3, 62,7% der Spiele gegen den MCTS-Solver zu gewinnen trotz einer durchschnittlich kleineren Anzahl von 400 Simulationen im Vergleich zu den 594 des MCTS-Solvers. MCTS-MB verbraucht einen Teil der Rechenzeit, um mit Hilfe von Alpha-Beta *proven wins* und *proven losses* zu propagieren. Der *Tradeoff* lohnt sich jedoch, da der so verbesserte Suchbaum zu einer besseren Spielstärke von MCTS-MB-3 im Vergleich zum MCTS-Solver führt.
Zudem fällt unter den drei Tiefen 3, 4 und 5, welche mehr Spiele gewonnen als verloren haben, ein leichter *Odd-Even Effect* auf: Die ungeraden Tiefen 3 und 5 weisen eine höhere Gewinnrate auf als die dazwischen liegende MCTS-MB-4. Das Ziel von Monte-Carlo-Tree-Search mit Alpha-Beta in der *Backpropagation* ist es, während der *Backpropagation* insbesondere *proven losses* für den Knoten zu finden, um so *sudden deaths* zu vermeiden. In MCTS-MB wird Alpha-Beta für die Kindknoten aufgerufen, somit endet eine ungerade Alpha-Beta-Suche; wie Tiefe 3 oder 5, auf dem Zug des Kindknotens. Da der Kindknoten für den Gegner spielt, ist es sinnvoller, auch auf einem Zug für den Kindknoten zu enden, da es sich um einen Zug handeln könnte, mit welchem der Gegner gewinnt. Findet MCTS-MB diesen Sieg für den Gegner, kann der Knoten als *proven loss* markiert werden. Somit ist es durchaus plausibel, dass MCTS-MB-3 und MCTS-MB-5 durch ungerade Tiefe in der Alpha-Beta-Suche eine höhere Spielstärke aufweisen als MCTS-MB-4.
Wie zu erwarten zeigt sich auch, dass mit steigender Tiefe der Alpha-Beta-Suche die Anzahl der Simulationen pro Zug abnehmen. Für die Tiefen 1 und 2 scheinen die gewonnenen Informationen durch die Alpha-Beta-Suche zu gering zu sein, um selbst den doch recht geringen Verlust an Simulationen pro Zug auszugleichen. Die Tiefen 3 bis 5 scheinen den *sweet spot* zu treffen, in dem die gewonnenen Informationen die Einbuße an Simulationen mehr als ausgleichen und zu einer erhöhten Spielstärke führen. Ab Tiefe 6 ist die reduzierte Anzahl an Simulationen zu groß, um durch die gewonnenen Informationen aus der Alpha-Beta-Suche die Spielstärke des Algorithmus zu stärken.
Wird die Zeit pro Zug auf 20 Sekunden erhöht, so fällt als erstes auf, dass der durch Alpha-Beta-Suche erreichte Vorteil an Relevanz verliert. Es sind nur noch MCTS-MB-3 und MCTS-MB-5 in der Lage, mehr Spiele gegen den MCTS-Solver zu gewinnen als zu verlieren. Da der Sinn von Alpha-Beta in der *Backpropagation* ist, als *fallback* für die *Backpropagation* von *proven wins* und *proven losses* innerhalb des MCTS-Solvers zu dienen, ist es also gut möglich, dass mit einer ausreichenden Anzahl von Simulationen, die durch Alpha-Beta gefundenen *proven wins* und *proven losses* in späteren Simulationen sowieso durch den normalen MCTS-Solver gefunden werden würden. Die verwendete Zeit für Alpha-Beta wird so überflüssig und sogar zum Nachteil für die meisten Suchtiefen. Da die Suchtiefen 3 und 5 weiterhin einen *sweet spot* treffen, scheint der so erhaltene Vorteil weiterhin auszureichen, um für diese Tiefen das Defizit an Simulationen auszugleichen.
Im Spiel gegen Alpha-Beta bestätigen sich erneut die Erkenntnisse aus dem Spiel gegen den MCTS-Solver. Zwar ist der Unterschied zwischen den Varianten kleiner und der Verlauf sehr ähnlich, jedoch sind MCTS-MB-3 und MCTS-MB-5 erneut die besten Varianten.

4.2.3 MCTS-MS

Bei der Nutzung von Alpha-Beta während der *Selection*-Phase mit nur 2 Sekunden pro Zug fällt ein Punkt deutlich auf: Suchtiefe 6 ist besser als Suchtiefe 4, und 4 wiederum ist besser als 2. Das Verhalten ähnelt also dem eines normalen Alpha-Beta-Algorithmus, bei dem höhere Suchtiefen zu stärkeren Spielen führen. Es liegt also nahe, dass MCTS-MS hier während der *Selection*-Phase mit Alpha-Beta einen Suchbaum aufbaut, der dem durch die Alpha-Beta-Suche vorgegebenen Baum sehr ähnelt. Die gewählten Züge sind wahrscheinlich den Zügen nahezu gleich, die von einem normalen Aufruf von Alpha-Beta mit der Suchtiefe zurückgegeben werden. Monte-Carlo-Tree-Search hat also nicht genug Simulationen, um eine eigenständige Statistik aufzubauen, wenn die *Selection* von Alpha-Beta dominiert wird.
Auch der Verlauf mit steigendem *Visit-Threshold* gibt dieser Annahme Gewicht, da hier der Einfluss von Alpha-Beta sinkt und somit die Siegrate der Varianten sich einander annähert.
Es wäre also sinnvoll, für MCTS-MS mit 2 Sekunden pro Zug eine weitere Analyse durchzuführen, in der man die gewählten Züge sowie den aufgebauten Suchbaum von MCTS-MS mit einer festen Alpha-Beta-Suche über den Verlauf mehrerer Partien vergleicht, um festzustellen, inwieweit hier Parallelen existieren. Dies würde jedoch den Rahmen dieser Arbeit übersteigen.

Mit der angestiegenen Anzahl an Simulationen pro Zug bei 20 Sekunden verändert sich das Verhalten von MCTS mit MiniMax *Selection* deutlich. Hier wird klar, wie der Algorithmus versucht, die Anzahl an Simulationen mit informierter *Selection* zu balancieren. Am besten zu erkennen ist dies an der Variante mit Suchtiefe 6: Für niedrige *Visit-Thresholds* wird die Alpha-Beta-Suche häufig eingesetzt, was zu einer niedrigen Anzahl von Simulationen führt. Somit kann keine gute MCTS-Statistik aufgebaut werden und die Siegrate ist entsprechend niedrig. Wird das *Visit-Threshold* erhöht und somit die Alpha-Beta-Suche nur selektiv eingesetzt, ist der Algorithmus in der Lage, eine gute MCTS-Statistik aufzubauen, und die durch die Alpha-Beta-Suche gefundenen *proven wins* und *proven losses* helfen dabei, eine Siegrate von über 50% zu erreichen.
So lässt sich auch der Verlauf von MCTS-MS mit Suchtiefe 4 erklären: Für ein niedriges *Visit-Threshold* wird die Alpha-Beta-Suche zu häufig aufgerufen und verhindert damit die benötigte Anzahl an Simulationen in der vorgegebenen Zeit. Mit Erhöhung des *Thresholds* findet der Algorithmus bei *Threshold* 20 einen *sweet spot*, in dem der Zeitaufwand zwischen Simulationen und Alpha-Beta-Suche so balanciert wird, dass die Siegrate von 57% erreicht werden kann. Mit weiterer Erhöhung des *Thresholds* sinkt der Einsatz der Alpha-Beta-Suche, sodass der erhaltene Vorteil minimal wird und damit die Siegrate auch wieder sinkt.
Für Suchtiefe 2 zeigt sich, dass die von Alpha-Beta erhaltenen Informationen so gering sind, dass sie nur einen Effekt haben, wenn die Alpha-Beta-Suche häufig aufgerufen wird wie für *Visit-Threshold* 1. Da der Zeitaufwand für einen Alpha-Beta-Suche mit Tiefe 2 im Vergleich gering ausfällt, ist der Algorithmus dennoch in der Lage, eine ausreichende Anzahl an Simulationen durchzuführen. MCTS-MS-2-1 erreicht so eine Siegrate von 53,3%.
Das Verhalten von MCTS-MS gegen Alpha-Beta mit fester Suchtiefe ähnelt dem Verhalten gegen MCTS-Solver und bestätigt somit die Erkenntnisse aus der Gegenüberstellung mit MCTS-Solver mit 20 Sekunden pro Zug.

4.2.4 Vergleich der Ergebnisse mit „MCTS-Minimax Hybrids“ [4]

Im Paper „MCTS-Minimax Hybrids“ von Hendrik Baier und Mark H. M. Winands [4] wurde gezeigt, dass im Spiel „Catch The Lion“ jede der MCTS-MiniMax-Hybridvarianten eine Verbesserung gegenüber MCTS-Solver darstellt. Die Ergebnisse dieser Arbeit können das Ergebnis bestätigen, denn es konnte gezeigt werden, dass Varianten der Hybrid-Algorithmen in der Lage sind, eine Siegrate von mehr als 50% gegen MCTS-Solver zu erreichen.[4]

Dennoch gibt es Unterschiede in den Ergebnissen:
Für MiniMax in der *Simulation*-Phase zeigt sich generell ein ähnlicher Verlauf der Siegrate, wobei Suchtiefe 1 als bester Algorithmus und Tiefe 4 als der Algorithmus mit der niedrigsten Siegrate identifiziert wurden. Allerdings konnte in dieser Arbeit nur Suchtiefe 1 eine Siegrate von mehr als 50% erreichen, im Gegensatz zu dem Ergebnis aus dem Paper, in dem Tiefe 1 bis 3 hohe Siegraten erzielen konnten.
Im Test von MiniMax in der *Backpropagation*-Phase zeigte sich ein ähnlicher Glockenkurvenverlauf über die Suchtiefe. Im Paper ist jedoch Suchtiefe 4 die beste Variante von MCTS-MB, was in dieser Arbeit mit Tiefe 3 und 5 als beste Variante nicht bestätigt werden konnte. Zudem haben alle Varianten im Paper eine Siegrate von über 50%, was ebenfalls nicht bestätigt werden konnte.
In den Experimenten mit MiniMax in der *Selection*-Phase konnte der Verlauf von Suchtiefe 6 sowie das Ansteigen und Absinken von Suchtiefe 4 mit dem *Visit-Threshold* bestätigt werden. Hier ist jedoch auch zu erkennen, dass die Siegraten im Paper deutlich höher sind als in dieser Arbeit.[4]
Die Unterschiede in den Ergebnissen zwischen dem Paper und dieser Arbeit lassen sich mit hoher Wahrscheinlichkeit auf technische Gründe zurückführen. Die Implementierung von Monte-Carlo-Tree-Search mit MiniMax erreicht im Paper je nach Variante zwischen 10.000 und 20.000 Simulationen pro Sekunde, während Monte-Carlo-Tree-Search in dieser Arbeit nur 10.000 Simulationen mit 20 Sekunden pro Zug nahe kommt.
Die Experimente in dieser Arbeit wurden auf einem möglicherweise weniger leistungsstarken Rechner durchgeführt. Zudem wurde mit Python in dieser Arbeit eine relativ ineffiziente Programmiersprache gewählt. Der größte Einfluss auf den Unterschied dürfte jedoch die Implementierung der Alpha-Beta-Suche und die damit verbundene Repräsentation des Spiels „Catch The Lion“ haben. Ist diese weniger effizient, so kostet die Ausführung von Alpha-Beta den Hybridalgorithmus mehr Zeit und verhindert eine höhere Anzahl an Simulationen und damit einhergehende Spielstärke. Somit ist wahrscheinlich eine effizientere Implementierung der Algorithmen und des Spiels in einer hardwarenahen Programmiersprache der Grund für die höhere Spielstärke der Hybridalgorithmen im Paper „MCTS-Minimax Hybrids“ von Hendrik Baier und Mark H. M. Winands [4].

Der Vergleich der Hybridalgorithmen anhand eines anderen Algorithmus als MCTS-Solver wird im Paper „MCTS-Minimax Hybrids“ [4] nicht diskutiert. Diese Arbeit setzt hier an und vergleicht die Hybriden mit der Alpha-Beta-Suche, um die erhöhte Spielstärke von MiniMax-MCTS-Hybriden auch anhand eines alternativen Algorithmus zu beweisen. Dies ist gelungen, da die Hybridalgorithmen auch im Spiel gegen Alpha-Beta das gleiche Verhalten aufweisen wie gegen MCTS-Solver. Somit konnte diese Arbeit auf dem Paper von Hendrik Baier und Mark H. M. Winands aufbauen und die erhöhte Spielstärke von MiniMax-MCTS-Hybriden auch in einem anderen Licht bestätigen.

4.3 MCTS-MiniMax-Hybriden gegen MiniMax

Aus den Ergebnissen der Gegenüberstellung von MTD(f) und den MCTS-MiniMax-Hybriden sticht ein Punkt eindeutig heraus: Die Alpha-Beta-Suche ist der bessere Algorithmus für „Catch The Lion“. Wenn man die Siegrate von unter 40%, welche die Hybriden gegen Alpha-Beta mit der festen Suchtiefe von 4 erzielen konnten, betrachtet, ist es ebenfalls keine Überraschung, dass der nicht künstlich eingeschränkte MTD(f)-Algorithmus mit einer Siegrate von mehr als 90% dominiert.
Betrachtet man „Catch The Lion“, ergibt es außerdem durchaus Sinn, dass Alpha-Beta eine hohe Spielstärke hat. „Catch The Lion“ ist mit einer hohen Anzahl von *sudden deaths* sowie einem relativ geringen *branching factor* für Alpha-Beta gemacht. Im Gegensatz dazu ist Monte-Carlo-Tree-Search in Spielen mit geringer Anzahl an *sudden deaths* und hohem *branching factor* wie „Go“ am besten.
Das Spiel „Catch The Lion“ dient somit auch dazu, zu zeigen, inwiefern Monte-Carlo-Tree-Search durch Alpha-Beta verbessert werden kann. Diese Verbesserung wurde in dieser Arbeit schon mehrfach gezeigt und taucht dementsprechend auch wieder in dieser Gegenüberstellung auf. Betrachtet man MCTS-Solver, so ist zu erkennen, dass für 2 Sekunden und 20 Sekunden pro Zug der Algorithmus ungefähr eine Siegrate von 2% erreicht. Die Hybridalgorithmen erreichen im Verhältnis eine weitaus höhere Siegrate; allem voran Monte-Carlo-Tree-Search mit MiniMax in der *Selection*-Phase mit 20 Sekunden pro Zug und einer Suchtiefe von 6 sowie einem *Visit-Threshold* von 200. Der Algorithmus erreicht eine Siegrate von 6.6%, was zwar nicht besonders hoch ist, aber im Vergleich zu MCTS-Solver eine deutliche Verbesserung von 200% darstellt. Die Hybridalgorithmen sind also besonders effektiv, wenn sie ausreichend Zeit pro Zug erhalten und so eine deutliche Verbesserung gegenüber dem Basis MCTS-Solver Algorithmus im Spiel „Catch The Lion“.

5 Fazit

Das Ziel dieser Arbeit war es, einerseits die Ergebnisse des Papers „MCTS-Minimax Hybrids“ von Hendrik Baier und Mark H. M. Winands [4] zu bestätigen, wonach MCTS-MiniMax-Hybriden eine Verbesserung des MCTS-Solver Algorithmus im Spiel „Catch the Lion“ darstellen. Andererseits sollte auch unter den betrachteten Algorithmen der beste für das Spiel „Catch the Lion“ ermittelt werden.
Es konnte erfolgreich bestätigt werden, dass MCTS-MiniMax-Hybriden eine Verbesserung des MCTS-Solver Algorithmus darstellen. Dabei wurde auch auf die Ergebnisse des Papers von Baier und Winands [4] aufgebaut, in dem die Hybridalgorithmen auch gegen Alpha-Beta getestet wurden. Die Experimente haben nicht nur die erhöhte Spielstärke der Hybriden aufgezeigt, sondern auch, aufgrund welcher interner Mechanismen bestimmte Varianten des Hybridalgorithmus besser gespielt haben als andere.
Es ist jedoch eindeutig aus den Ergebnissen hervorgegangen, dass die Alpha-Beta-Suche dennoch der bessere Algorithmus für das Spiel „Catch the Lion“ ist. Die beste Variante des MiniMax-Algorithmus, MTD(f), war in der Lage, mehr als 90% der Spiele gegen jede Variante von Monte-Carlo-Tree-Search zu gewinnen.
Monte-Carlo-Tree-Search ist exzellent in Spielen mit hohem *branching factor* wie „Go“. In dieser Arbeit ist jedoch klar geworden, dass für den Einsatz von Monte-Carlo-Tree-Search in Spielen, die auch eine hohe Anzahl an *sudden deaths* aufweisen, MCTS definitiv im Zusammenhang mit Alpha-Beta verwendet werden sollte. Hat das Spiel allerdings einen relativ geringen *branching factor* wie „Catch the Lion“, so ist der MCTS-MiniMax-Hybrid eine Option, aber die Alpha-Beta-Suche bleibt die bessere Wahl.

Es sind außerdem weitere Forschungsfragen entstanden, die zukünftig betrachtet werden sollten:
Für MiniMax in der *Simulation*-Phase wäre es interessant zu beachten, inwiefern der Erfolg von Suchtiefe 1 mit der Alpha-Beta-Mechanik korreliert oder doch nur im Zusammenhang mit der Heuristik steht. Außerdem wäre es für MCTS-MR interessant zu untersuchen, inwieweit sich eine verbesserte Alpha-Beta-Suche durch eine erweiterte Evaluation auf die Spielstärke des Algorithmus auswirkt.
Im Test von MCTS-MS mit 2 Sekunden pro Zug ist ein sehr ähnliches Verhalten zur normalen Alpha-Beta-Suche aufgefallen. Inwiefern hier Parallelen bestehen könnten, sollte ebenfalls untersucht werden.
Der beste Algorithmus für „Catch the Lion“, MTD(f), hat in den Tests nur eine minimale Verbesserung zu Alpha-Beta mit Transpositionstabellen aufgewiesen, obwohl aus den Tests zur Zeit pro Tiefe eine größere Verbesserung zu erwarten wäre. Hier sollten die Algorithmen anhand unterschiedlicher Zeiten pro Zug sowie gegen andere Gegner weiter verglichen werden.

Um abschließend noch einmal auf das Beispiel aus der Einleitung einzugehen: Hätte Google DeepMinds AlphaZero den Top-Go-Spieler Lee Sedol 5-0 anstatt 4-1 besiegen können, wenn der Monte-Carlo-Tree-Search-Algorithmus mit der Alpha-Beta-Suche verbessert worden wäre?
Die Antwort auf diese Frage ist nicht so leicht zu finden. In dieser Arbeit wurde gezeigt, dass durch die Einführung der Alpha-Beta-Suche in die Monte-Carlo-Tree-Search der Algorithmus erfolgreicher in Spielen mit *sudden deaths* wird. Stimmt es also tatsächlich, dass der Grund für die Niederlage von AlphaZero in dieser einen Partie eine versteckte Niederlage in einer unauffälligen Zugfolge war, dann ist es gut möglich, dass ein Algorithmus, der besser im Finden dieser *sudden deaths* ist, die Niederlage erkannt und hätte umgehen können.
Andererseits muss jeder MCTS-MiniMax-Hybrid die Anzahl der Simulationen mit der Suchtiefe von Alpha-Beta balancieren. Es ist nicht klar, ob eine erfolgreiche Balance dieser im Spiel „Go“ möglich ist.
Trotz allem wäre ein Rematch von Lee Sedol mit einem MCTS-MiniMax-Hybrid gegen AlphaZero eine durchaus interessante Partie.

ChatGPT

ChatGPT 3.5 wurde in dieser Arbeit zur Korrektur verwendet. Dabei passte das AI-Tool die Gramatik und Rechtschreibung an, der Inhalt blieb jedoch unverändert.
URL: https://chat.openai.com

Literaturverzeichnis

[1] Bruce Abramson. *The expected-outcome model of two-player games*. Morgan Kaufmann, 2014.

[2] Alphago versus lee sedol. https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol#. Accessed: 9.02.2024.

[3] Alphazero. https://www.chessprogramming.org/AlphaZero. Accessed: 31.01.2024.

[4] Hendrik Baier and Mark HM Winands. Mcts-minimax hybrids. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):167–179, 2014.

[5] Hendrik Baier and Mark HM Winands. Mcts-minimax hybrids with state evaluations. *Journal of Artificial Intelligence Research*, 62:193–231, 2018.

[6] Bitboards. https://www.chessprogramming.org/Bitboards. Accessed: 19.01.2024.

[7] Guillaume M Jb Chaslot, Mark HM Winands, H Jaap van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[8] Computer go. https://en.wikipedia.org/wiki/Computer_Go. Accessed: 9.02.2024.

[9] Dōbutsu shōgi. https://en.wikipedia.org/wiki/DÅ■butsu_shÅ■gi. Accessed: 18.01.2024.

[10] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280, 2007.

[11] Iterative deepening. https://www.chessprogramming.org/Iterative_Deepening. Accessed: 31.01.2024.

[12] Iterative deepening depth-first search. https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search. Accessed: 8.04.2024.

[13] Mailbox. https://www.chessprogramming.org/Mailbox. Accessed: 19.01.2024.

[14] Mo. https://www.chessprogramming.org/Move_Ordering. Accessed: 8.04.2024.

[15] Monte carlo tree search. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. Accessed: 9.02.2024.

[16] Mtd(f). https://en.wikipedia.org/wiki/MTD(f). Accessed: 8.02.2024.

[17] Nw. https://www.chessprogramming.org/Null_Window. Accessed: 7.04.2024.

[18] Aske Plaat. Mtd(f) algorithm, a minimax algorithm faster than negascout. https://askeplaat.wordpress.com/534-2/mtdf-algorithm/. Accessed: 31.01.2024.

[19] Alexander Reinefeld. Nullfenster-suche. https://gi.de/informatiklexikon/nullfenster-suche. Accessed: 7.02.2024.

[20] Transposition table. https://www.chessprogramming.org/Transposition_Table. Accessed: 31.01.2024.

[21] Uct. https://www.chessprogramming.org/UCT. Accessed: 22.02.2024.

[22] Benjamin Blankertz und Vera Roehr. Algorithmen und datenstrukturen. 2020, TU Berlin, Algorithmen und Datenstrukturen.

[23] Mark HM Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver, 2008.

[24] Mark HM Winands and N Lee. Monte-carlo tree search., 2019.

[25] Zobrist hashing. https://www.chessprogramming.org/Zobrist_Hashing. Accessed: 31.01.2024.

6 Anhang

6.0.1 Repositories

Code Repository on GitHub: CatchTheLion
URL: https://github.com/bruno-sfr/CatchTheLionRemade

Documentation of the hydra cluster: Hydra
URL: https://git.tu-berlin.de/ml-group/hydra/documentation

6.0.2 Grafiken

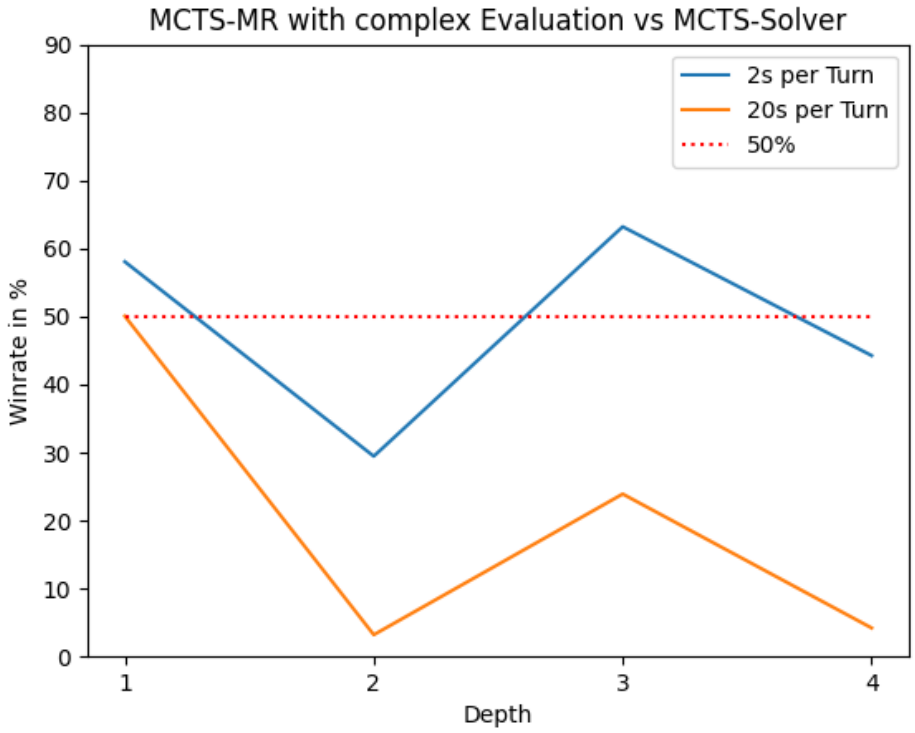


Abbildung 6.1: MCTS-MR mit erweiterte Evaluation gegen MCTS-Solver
(Quelle: eigene Abbildung)

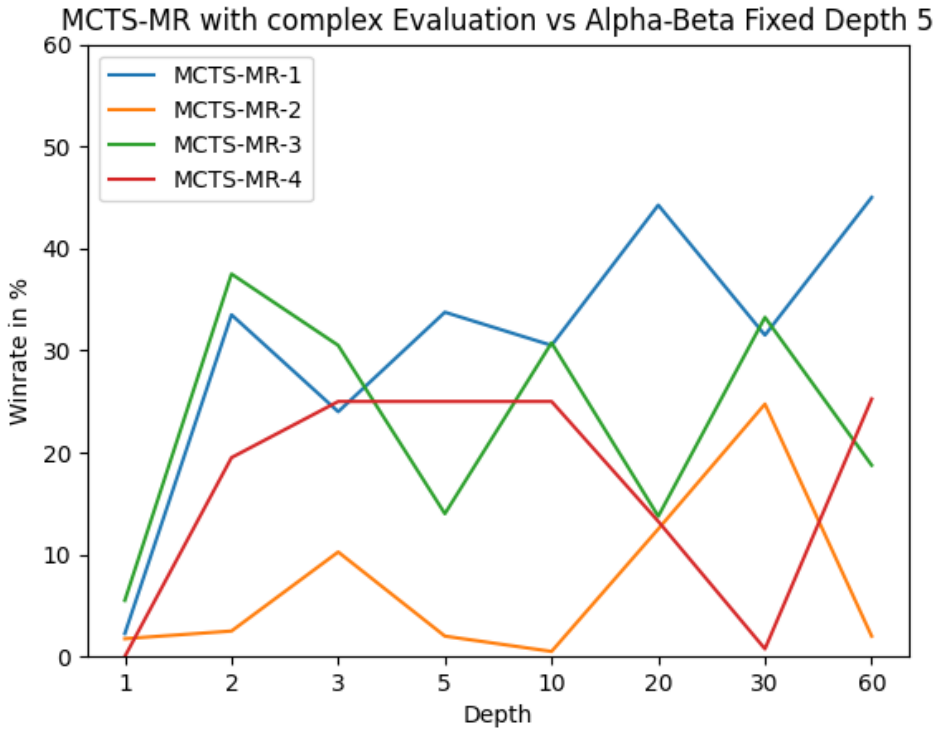


Abbildung 6.2: MCTS-MR mit erweiterte Evaluation gegen Alpha-Beta (Quelle: eigene Abbildung)

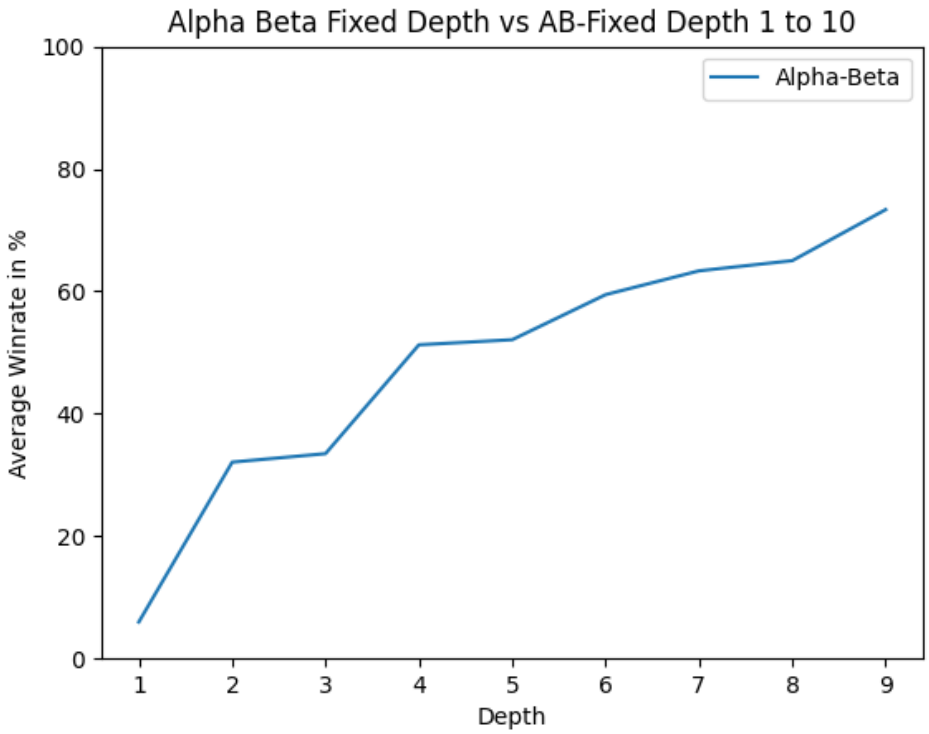


Abbildung 6.3: Alpha-Beta mit fester Tiefe gegen Alpha-Beta Tiefe 1 bis 10
(Quelle: eigene Abbildung)