Systematischer Vergleich von Alpha-Beta und Monte Carlo Tree Search Varianten in dem Spiel Yavalath

Bachelorarbeit

Jan Lennart Tamm
456810

28. Mai 2025

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

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

Kurzfassung

Die Alpha-Beta-Suche und die Monte Carlo Tree Search (MCTS) sind weit verbreitete Methoden im Bereich der algorithmischen Zugfindung für Spiele und können bei einer Vielzahl von Spielen eingesetzt werden. Dabei bieten sie unterschiedliche Eigenschaften, die je nach Anwendungsbereich ihre Effektivität beeinflussen. Im Rahmen dieser Arbeit werden die beiden Algorithmen, einschließlich hybrider MCTS-Varianten mit der Alpha-Beta-Suche und der Proof Number Search, im Kontext des Spiels Yavalath untersucht.

Ziel ist es, durch einen systematischen Vergleich der Suchalgorithmen, ihre Anwendbarkeit und Grenzen für Yavalath aufzuzeigen. Dafür werden in dieser Arbeit die theoretischen Grundlagen erläutert und die Implementierungen der Methoden durch Spielsimulationen evaluiert.

Die Ergebnisse zeigen, dass die Alpha-Beta-Suche in Yavalath klare Vorteile gegenüber MCTS aufweist. Die Beobachtungen über die hybriden MCTS-Ansätze legen nahe, dass diese erfolgreich als Optimierungen der Methode in Yavalath eingesetzt werden können.

Inhaltsverzeichnis

1 Einleitung 1
- 1.1 Das Spiel Yavalath . . . 1
- 1.2 Ziel der Arbeit . . . 1
- 1.3 Literatur . . . 2
- 1.4 Struktur der Arbeit . . . 3

2 Methoden 4
- 2.1 Bitboards . . . 4
- 2.2 Alpha-Beta-Suche . . . 6
  - 2.2.1 Bewertungsfunktion . . . 8
  - 2.2.2 Iterative Tiefensuche . . . 8
  - 2.2.3 Transpositionstabelle . . . 9
- 2.3 Monte Carlo Tree Search . . . 10
  - 2.3.1 Upper Confidence Bounds angewandt auf Bäume (UCT) . . . 13
  - 2.3.2 Simulation mit General Domain Knowledge . . . 14
  - 2.3.3 Eigenschaften von MCTS . . . 14
  - 2.3.4 MCTS in Kombination mit Alpha-Beta-Suche . . . 14
  - 2.3.5 MCTS in Kombination mit Poof Number Search . . . 16

3 Ergebnisse 18
- 3.1 Berechnungszeit Alpha-Beta-Suche . . . 18
- 3.2 Wahl der MCTS Konfiguration . . . 20
- 3.3 Vergleich der ausgewählten Algorithmen . . . 23
  - 3.3.1 Anzahl der Iterationen . . . 24
  - 3.3.2 Untersuchte Knoten . . . 24
  - 3.3.3 Speicherbedarf der Transpositionstabelle . . . 25
  - 3.3.4 Ergebnisse der Spiele . . . 25

4 Diskussion 29
- 4.1 Vergleich der Alpha-Beta Varianten . . . 29
- 4.2 Vergleich der MCTS Varianten . . . 29
  - 4.2.1 Explorationskonstante c . . . 29
  - 4.2.2 General Domain Knowledge . . . 30
- 4.3 Vergleich der ausgewählten Algorithmen . . . 30
- 4.4 Limitationen . . . 31
- 4.5 Beitrag der Arbeit . . . 31

5 Fazit 32

Abbildungsverzeichnis

1.1 Beispiel einer Spielposition. Schwarz ist am Zug und muss den drohenden Sieg von Weiß zu blockieren, verliert jedoch unmittelbar durch diesen Zug. 1

2.1 Visualisierungen der Bitpositionen eines Bitboards; Separierungsbits sind blau markiert. 4
2.2 Beispiel einer Spielsituation nach 11 Zügen; Schwarz ist an der Reihe. 5
2.4 Vorteilhaftes Dreiecksmuster 8
2.5 Beispiel für die Berechnung von Zobrist Hashwerten 10
2.6 Eine Iteration des Monte Carlo Tree Search. Nach [5] 11
2.7 Flussdiagram von MCTS mit Alpha-Beta nach [12] 15
2.8 PNS Suchbaum mit proof und disproof numbers nach [6] 16

3.1 Relative druchschnittliche Berechnungszeit der Alpha-Beta Varianten pro Suchtiefe und Algorithmus 19
3.2 Laufzeiten der Alpha-Beta Varianten pro Spielposition (in Sekunden) 19
3.3 Testschritt 1: Durchschnittliche Siegesrate der c-Werte im Spiel untereinander ohne GDK (Zugzeit 1 Sekunde) 20
3.4 Testschritt 2: Siegesrate der weiter untersuchten c-Werte (Zugzeit 10 Sekunden) . . . 21
3.5 Testschritt 3: Siegesraten für **MCTS** ohne GDK gegen **MCTS** mit GDK und jeweils $c = 0.5$ (Zugzeit 10 Sekunden) 21
3.6 Testschritt 4: Durchschnittliche Siegesrate der c-Werte im Spiel untereinander mit GDK (Zugzeit 1 Sekunde) 22
3.7 Testschritt 5: Durchschnittliche Siegesrate der c-Werte im Spiel untereinander mit GDK (Zugzeit 5 Sekunden) 23
3.8 Durchschnittliche Iterationen der MCTS Varianten pro Zug 24
3.9 Durchschnittliche Anzahl an untersuchten Knoten pro Zug 25
3.10 Siegesraten der Algorithmen im Round-Robin Verfahren 27
3.11 Spielstellungen für die Tests 28

Tabellenverzeichnis

2.1 Bewertungen der einzelnen Muster durch die Bewertungsfunktion für Alpha-Beta . . 8

3.1 Durchschnittliche Berechnungszeit der Alpha-Beta Varianten je Suchtiefe (in Sekunden) 18
3.2 Testschritt 4: Siegesrate (in %, gerundet) der untersuchten c-Werte für **MCTS** mit GDK (Zugzeit 1 Sekunde). Unentschieden werden als nicht-Sieg für beide gewertet. . 22
3.3 Testschritt 5: Siegesrate (in %, gerundet) der untersuchten c-Werte für **MCTS** mit GDK (Zugzeit 5 Sekunden). Unentschieden werden als nicht-Sieg für beide gewertet. 23
3.4 Siegesraten (in %, gerundet). Teil 1 26
3.5 Siegesraten (in %, gerundet). Teil 2 26
3.6 Anzahl und Anteil der Spiele, welche in einem Unentschieden endeten. 26

1 Einleitung

1.1 Das Spiel Yavalath

Yavalath ist ein strategisches Spiel für zwei Personen. Gespielt wird auf einem hexagonalen Spielbrett mit 61 sechseckigen Feldern. Die Spieler platzieren abwechselnd Spielsteine auf freie Felde mit dem Ziel, als Erster vier der eigenen Steine in einer durchgehenden Reihe anzuordnen. Wer allerdings eine Reihe aus nur drei Steinen bildet, verliert das Spiel. Das Spiel endet in einem Unentschieden, wenn alle Felder besetzt sind und noch kein Spieler die Partie gewonnen hat. Eine Reihe kann aufgrund der Form der Felder nur diagonal oder horizontal gebildet werden. Bereits gesetzte Steine werden zu keinem Zeitpunkt des Spiels wieder entfernt.

Das Spiel wurde von Browne [2] in seiner Dissertation vorgestellt. Es entstammt einer Sammlung von Spielen, deren Spielregeln durch einen evolutionären Algorithmus aus bereits bekannten Spielen generiert wurden.

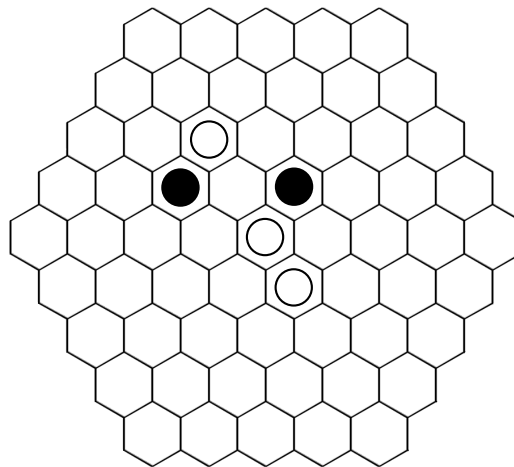


Abbildung 1.1: Beispiel einer Spielposition. Schwarz ist am Zug und muss den drohenden Sieg von Weiß zu blockieren, verliert jedoch unmittelbar durch diesen Zug.

1.2 Ziel der Arbeit

Aufgrund der hohen Anzahl möglicher Züge ist das effiziente Finden siegbringender Züge eine Herausforderung. Zwar nimmt die Anzahl der möglichen Züge mit der Länge des Spiels konstant ab, doch können Spiele sehr schnell enden. Da eine Reihe aus drei Steinen eine Niederlage bedeutet, kann das

Bilden einer kleinen Reihe aus zwei Steinen[1] dem Gegner eine mögliche Angriffsfläche bieten. In ähnlichen Spielen wie Gomoku kann ein Zug, der eine kurze Reihe bildet, als ein direkter Schritt in Richtung einer Siegesreihe gesehen werden. Für Yavalath ist das aus dem genannten Grund allerdings nicht der Fall. Somit muss neben dem Setzen einer eigenen Siegesreihe und dem Blockieren der Reihen des Gegners auch beachtet werden, dem Gegner keine Angriffsmöglichkeit auf eigene kleinere Reihen zu bieten.
Zusätzlich ist Yavalath in der wissenschaftlichen Literatur weitaus weniger präsent als andere Spiele wie beispielsweise Schach oder Gomoku. Deshalb ist das vorhandene domänenspezifische Wissen über Yavalath vergleichsweise gering.
In dieser Arbeit werden zwei algorithmische Ansätze zur Zugfindung in Spielen untersucht: die Alpha-Beta-Suche und die Monte Carlo Tree Search. Beide bringen Vor- und Nachteile mit sich, die im Kontext von Yavalath jeweils gegensätzlich ausfallen.
Während die Alpha-Beta-Suche Wissen über das Spiel für eine Bewertungsfunktion benötigt, kommt die Monte Carlo Tree Search in ihrer Grundform gänzlich ohne Vorwissen aus.
Im Gegenzug stellt das plötzliche Spielende nach einer Zugfolge mit erzwungenen Zügen, zu der wenige spezifische Züge führen, eine Herausforderung für die Monte Carlo Tree Search dar [12] [6]. Denn in solchen Fällen kann die Wahrscheinlichkeit, auf den richtigen Zug zu konvergieren, drastisch abnehmen. Daher werden in dieser Arbeit zusätzlich Modifkationen der Monte Carlo Tree Search, wie von Li [12] die Kombination mit der Alpha-Beta-Suche und von Doe [6] mit der Proof Number Search vorgeschlagen, untersucht.
Das Ziel dieser Arbeit ist ist es, eine für Yavalath geeigneten Methode zu bestimmen, die mit geringem domänenspezifischem Wissen in der Lage ist, als ein starker Spieler zu agieren.

1.3 Literatur

Die Literatur über das Spiel Yavalath beschränkt sich hauptsächlich auf Veröffentlichungen von Cameron Browne. Insbesondere stellte er Yavalath erstmals als Teil seiner Doktorarbeit [2] vor. In dem Anhang seiner Dissertation werden grundlegende Beobachtungen aus Partien erfahrener Spieler beschrieben. Außerdem schlägt er einen MCTS-Ansatz vor, dessen Rollouts mithilfe von generellem Domänenwissen über Brettspiele abwickelt.

Der Einsatz von Algorithmen für andere Brettspiele lässt sich auf den Kontext von Yavlath übertragen. Besonders aufschlussreich sind dabei Publikationen, die sich mit ähnliches Spielen wie Gomoku befassen. Aber auch Untersuchungen im Kontext von anderen gut erforschten Spielen, wie etwa Schach, lassen sich auf Yavalath übertragen.

Die Alpha-Beta-Suche ist einer der ersten Algorithmen im Bereich der künstlichen Intelligenz und wurde bereits 1961 von Edwards und Hart [7] beschrieben. Er wurde jedoch schon in den 1950er-Jahren von verschiedenen Personen und Forschungsgruppen unabhängig voneinander vorgeschlagen [13]. Korf [11] zeigte 1985 die Optimalität der iterativen Tiefensuche gegenüber des Brute-Force Ansatzes mit statischer Tiefe hinsichtlich der asymptotischen Laufzeit und Speicherplatzbedarfes. Außerdem argumentierte er, dass sich die Technik auf Algorithmen für Zwei-Personen-Spiele, wie

[1] Damit sind zwei Steine gemeint, die entweder direkt nebeneinander liegen oder durch ein Feld von einander entfernt sind.

die Alpha-Beta-Suche, übertragen lässt. Die Verwendung von Transpositionstabellen im Kontext der Alpha-Beta-Suche wurde von Greenblatt et al. [8] beschrieben und von Breuker et al. [1] experimentell weiter untersucht. Dabei kommt die von Zobrist [16] vorgestellte Hashing-Methode zum Einsatz.

Monte Carlo Tree Search wird ebenfalls in zahlreichen Veröffentlichungen untersucht. Die Werke von Kocsis und Szepesvári [10] sowie Chaslot et al. [5] gehören zu den ersten, die diese Methode beschrieben, während Browne et al. [4] eine Übersicht über die 2012 bereits veröffentlichte Literatur zu dem Thema gaben.
Li et al. [12] formalisierten die Problematik von plötzlichen Spielenden, als sogenannte sudden death Situationen, bzw. sudden win Situationen, und stellten einen Ansatz vor, um die Nachteile dieser Herausforderung zu minimieren. Dabei wird die Monte Carlo Tree Search in Kombination mit der Alpha-Beta-Suche einsetzt. Die Veröffentlichung beschrieb diese Konzepte im Kontext des Yavalath-ähnlichen Spiels Gomoku. Außerdem stellten Doe et al.[6] einen Ansatz zur Bewältigung der sudden death/win-Problematik durch die Kombination mit Proof Number Search vor, welche auf domänenspezifisches Wissen verzichtet.

1.4 Struktur der Arbeit

In dieser Arbeit werden als erstes die verwendeten Methoden vorgestellt:

- **Bitboards**: Eine effiziente Darstellung von Spielbrettern.
- **Alpha-Beta-Suche**: Ein Algorithmus, der Spielstellungen mittels einer Bewertungsfunktion bewertet und einen Zug sucht, welcher innerhalb einer limitierten Suchtiefe die beste Spielstellung garantiert.
- **Monte Carlo Tree Search**: Ein von domänenspezifischem Wissen unabhängiger Algorithmus, der durch die Simulation von großen Menge an Spielen einen aussichtsreichen Zug bestimmt.

Zusätzlich werden weitere Varianten der beiden Suchalgorithmen vorgestellt.

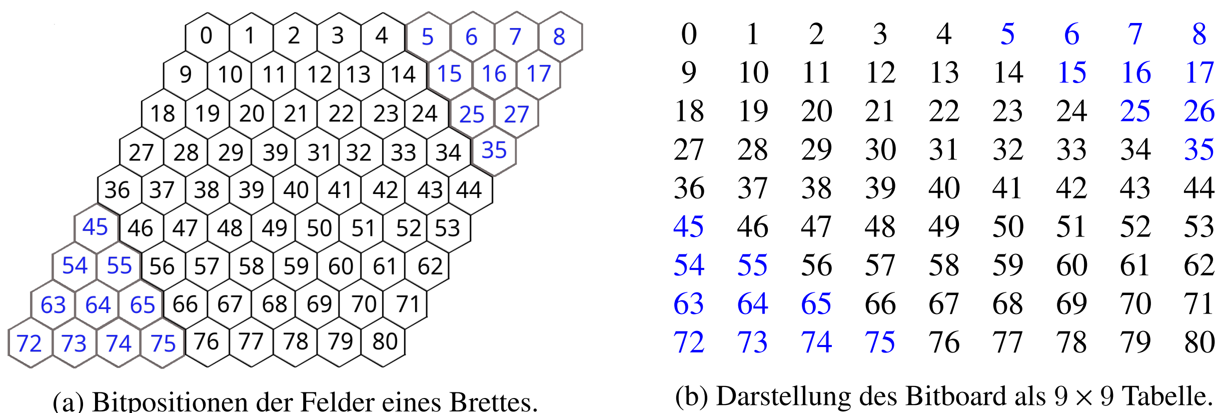
Im nächsten Abschnitt wird dann experimentell die effizienteste Implementierung der Alpha-Beta-Suche und die vielversprechendste Konfiguration des Basis-MCTS bestimmt. Danach werden die ausgewählten Variationen untereinander und mit den erweiterten Monte Carlo Tree Search Varianten verglichen.

In dem darauffolgenden Kapitel wird ein Zusammenhang zur bereits existierenden Literatur hergestellt, um die Ergebnisse der Experimente mit den Erwartungen abzugleichen. Zuletzt wird auf die Limitationen der Arbeit eingegangen und ein Ausblick auf eine mögliche Fortsetzung des untersuchten Sachverhaltes gegeben.

2 Methoden

2.1 Bitboards

Die hier vorgestellte Verwendung von Bitboards basiert auf den Methoden, wie sie von Herzberg [9] für das Spiel Connect Four beschrieben wurden. Die Anpassung an Yavalath erfolgte durch die Konzepte für Bitboards mit zugrunde liegnden hexagonalen Spielfeldern, wie sie von Browne [3] in 2014 vorgestellt wurden.
Ein Bitboard ist eine Darstellungsweise für Spielbretter, in der das gesamte Brett als eine einzige Zahl repräsentiert wird. Jedem Feld wird dabei ein Bit im Bitboard zugeordnet, das angibt, ob dort bereits ein Spielstein gesetzt wurde oder noch nicht [3]. Da die Platzierungen der Steine beider Spieler gespeichert werden müssen, sind mindestens zwei Bitboards notwendig, um eine Spielsituation darzustellen. Es genügt ein Bitboard zu verwenden, das nur die Steine eines Spielers abbildet, und ein weiteres, das alle insgesamt gesetzten Steine enthält. Aus diesen Informationen können auch die Positionen der Steine des anderen Spieler abgeleitet werden. Es werden nicht mehr Bitboards verwendet, um die Nutzung des Speicherplatzes zu reduzieren. Um die Bitboards sinnvoll nutzen zu können, werden die Zeilen des Brettes durch Separierungsbits voneinander getrennt. Aufgrund des hexagonalen Struktur des Spielbrettes ist es für Yavalath sinnvoll die Separierungsbits wie in Abb. 2.1 dargestellt zu platzieren. Dies wird so gewählt, da so die Bits vertikal in Einer-, entlang der einen Diagonale in Neuner- und entlang der anderen Diagonale in Zehnerschritten angeordnet sind. Aus Abb. 2.1a lässt sich eine vereinfachte Darstellung eines Bitboards als 9×9 Tabelle ableiten.



(a) Bitpositionen der Felder eines Brettes.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

(b) Darstellung des Bitboard als 9×9 Tabelle.

Abbildung 2.1: Visualisierungen der Bitpositionen eines Bitboards; Separierungsbits sind blau markiert.

Die Verwendung von Separierungsbits bedeutet allerdings, dass ein Bitboard 81 Bits statt 61 benötigt. Eine von Programmiersprachen typischerweise verwendete 64-Bit Zahl reicht also nicht aus um ein

ganzes Bitboard darzustellen. Hier erweist sich die Wahl von Python von Vorteil, da in Python 3 Zahlen eine unbegrenzte Präzision haben[1]. Dadurch is es nicht erforderlich, die Bitboards auf mehrere Integer aufzuteilen.

Um eine Spielsituation darzustellen werden zwei Bitboards verwendet:

- *full* enthält die Informationen für alle gesetzten Steine.
- *current* speichert die von dem Spieler, welcher als nächstes am Zug ist, belegten Felder.

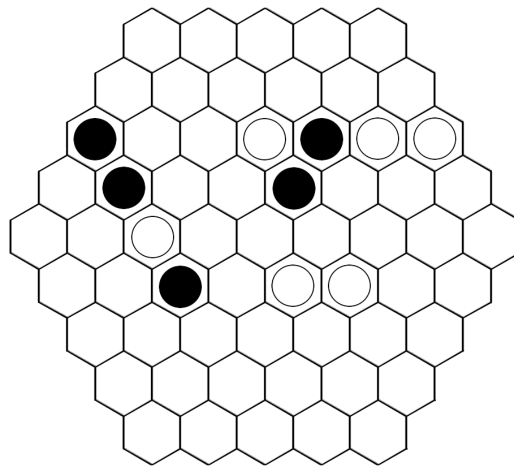


Abbildung 2.2: Beispiel einer Spielsituation nach 11 Zügen; Schwarz ist an der Reihe.

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(a) Bitboard *current* für Abb. 2.2

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	0	0
0	1	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	1	1	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(b) Bitboard *full* für Abb. 2.2

Um einen Zug auszuführen wird als erstes *current* durch eine bitweise XOR-Operation mit *full* aktualisiert. Dadurch repräsentiert *current* dann die Steine des Gegners. Anschließend wird dann in *full* das Bit auf eins gesetzt, das dem gespielten Zuges entspricht.

Nach jedem Zug wird das Brett auf Spielende überprüft. Dank der oben beschriebenen Wahl von Separierungsbits funktioniert dies für alle drei Achsen auf ähnliche Weise. Es wird dafür nur das Bitboard des Spielers betrachtet, der den letzten Zug ausgeführt hat. Dies wird im Folgenden als *board*

[1] `https://docs.python.org/3/library/stdtypes.html#typesnumeric` (Zugriff am 6.5.2025)

bezeichnet. Für die Gewinnbedingung in der Vertikalen wird *board* dreimal mit sich selbst bitweise UND-verknüpft, wobei jeweils ein Bitshift um eins, zwei bzw. drei vorgenommen wird. Ist die daraus resultierende Zahl nicht gleich Null, so befindet sich irgendwo vier Bits vertikal in einer Reihe. Zur Überprüfung auf eine Niederlage durch drei Steine in einer Reihe wird dies wiederholt, nur dass die UND-Verknüfung mit dem Bitshift um drei Bits nicht vorgenommen wird. Dank der oben beschriebenen Wahl von Separierungsbits funktioniert dies für die Diagonalen analog. Dabei werden dann die Bitshifts nicht in Einerschritten, sondern in Neuner- beziehungsweise Zehnerschritten durchgeführt. Ist kein Sieg oder Niederlage erkannt worden, so wird der Zugzähler, welcher nach jedem Zug um eines inkrementiert wird, überprüft. Ist dieser bei der maximalen Anzahl von 61 Zügen, endet die Partie in einem Unentschieden. Alternativ könnte *full* auch mit einem vollen Bitboard verglichen werden, um ein Unentschieden fest zustellen.

In Algorithmus 1 wird die Überprüfung auf das Spielende skizziert. Dabei bezeichnet >> ein Bitshift nach rechts und $\&$ eine bitweise UND-Verknüpfung. Die Richtungen d sind die vertikale und die beiden diagonalen Achsen und shift_d gibt die Schrittweite im Bitboard an, um in Richtung d auf dem Brett zum nächsten Feld zu gelangen. Für die Vertikale ist shift_d also 1, für die Diagonalen 9 bzw. 10.

Algorithmus 1 Überprüfung auf Spielende

```
 1: for each direction d do
 2:     win ← board & (board >> shift_d) & (board >> 2 · shift_d) & (board >> 3 · shift_d)
 3:     lose ← board & (board >> shift_d) & (board >> 2 · shift_d)
 4:     if win ≠ 0 then
 5:         return 1        ▹ Der Spieler der als letztes einen Stein gesetzt hat, gewinnt.
 6:     else if lose ≠ 0 then
 7:         return −1       ▹ Der Spieler der als letztes einen Stein gesetzt hat, verliert.
 8:     end if
 9: end for
10: if Zugzähler = 61 then
11:     return 0.5          ▹ Die Partie endet in einem Unentschieden.
12: else
13:     return 0            ▹ Das Spiel ist noch nicht vorbei.
14: end if
```

2.2 Alpha-Beta-Suche

Die Alpha-Beta-Suche (engl. Alpha-Beta Pruning) ist ein auf dem MiniMax-Verfahren aufbauender Suchalgorithmus für Zwei-Spieler-Spiele. In dem Verfahren agiert der Spieler, welcher am Zug ist, als maximierender und sein Gegner als minimierender Spieler. Es wird ein Suchbaum aufgebaut in dem die Knoten die Züge repräsentieren. Nach einer gegebenen Tiefe wird die aus den Zügen entstehende Spielsituation bewertet. Der Algorithmus ermittelt den Zug für den Max-Spieler, welcher ihn zur bestbewerteten Spielsituation bringt, welche unabhängig von den Zügen des Min-Spielers zu erreichen ist.
Die Alpha-Beta-Suche verbessert MiniMax indem Züge, die nicht betrachtet werden müssen, identi-

fiziert werden und an diesen Stellen ein Beschneidung des Suchbaums vorgenommen wird, genannt alpha-beta cutoff [7]. So kann die Berechnungszeit des Algorithmus verkürzt werden[2].

Für die Alpha-Beta-Suche wird zusätzlich ein Intervall $[\alpha, \beta]$ eingeführt. α ist der bisher höchste Wert, den der maximierende Spieler sicher erreichen kann. β ist analog der niedrigste Wert, welcher vom minimierenden Spieler garantiert erreicht werden kann. Initialisiert wird das Intervall mit $\alpha = -\infty, \beta = \infty$. α wird an Knoten des Max-Spielers aktualisiert, wenn ein Kindknoten einen Wert hat, der größer als α ist, und β an Knoten des Min-Spielers, wenn ein Kind einen kleineren Wert hat. Ein neues Kind eines Knotens erhält dessen aktuelles Alpha-Beta-Intervall. Falls an einem Knoten $\beta \leq \alpha$ gilt, wird ein alpha-beta cutoff ausgeführt und alle weiteren noch unerforschten Kinder müssen nicht betrachtet werden, da einem der Spieler bereits ein besserer Wert garantiert ist.

Aufgrund der Beobachtung, dass beim Tauschen der Vorzeichen der Bewertungen sich auch die Rolle als Maximierer und Minimierer vertauscht, kann eine Variante der Alpha-Beta-Suche beschrieben werden, die für beide Spieler gleich aussieht und Negamax genannt wird [13]. Dafür werden für Werte, welche zurückgegeben werden, die Vorzeichen umgedreht. Bei der Übergabe des Alpha-Beta-Intervalls werden die Werte getauscht und deren Vorzeichen umgedreht. Mit diesen Änderungen der Übergabe von Werten, genügt es dann einen Maximierer zu implementieren.

Algorithmus 2 Negamax Variante der Alpha-Beta-Suche nach [13]

```
 1: function NEGAMAX(state, depth, alpha, beta)
 2:     if depth = 0 or ISTERMINALSTATE(state) is true then
 3:         return EVALUATE(state)
 4:     end if
 5:     bestValue ← −∞
 6:     for all moves do
 7:         DOMOVE(state, move)
 8:         score ← −NEGAMAX(state, depth−1, −beta, −alpha)
 9:         UNDOMOVE(state, move)
10:         if score > bestValue then
11:             bestValue ← score
12:             if score > alpha then
13:                 alpha ← score
14:             end if
15:         end if
16:         if score ≥ beta then
17:             return bestValue
18:         end if
19:     end for
20:     return bestValue
21: end function
```

[2] Im Worst-Case ist dies allerdings nicht garantiert.

2.2.1 Bewertungsfunktion

Die Wahl der Bewertungsfunktion ist entscheidend für die Qualität der Züge, die von der Alpha-Beta-Suche gefunden werden. Grundlegend wird durch die Bewertungsfunktion nach Mustern in der Spielstellung gesucht. Dies wird ähnlich wie die Überprüfung auf Spielende aus Algorithmus 1 realisiert. Diese Muster erhalten Werte, welche aufsummiert werden. Um die Muster des Gegners mit in die Bewertung einfließen zu lassen, werden auch für diese die Werte aufsummiert und am Ende von der eigenen Bewertung subtrahiert.

Wie von Browne [2] beschrieben, stellen nicht blockierte Reihen aus zwei Steinen mit einer Lücke von zwei Feldern eine Gefahr für den Gegner dar, während zwei Steine ohne Lücke oder mit nur einem freien Feld dem Gegner eine Angriffsfläche bieten. Bemerkenswert ist, dass das Bilden dreieckiger Strukturen eine gute Strategie zu sein scheint. Wurden die Bewertungen der Muster daher so gewählt, dass Reihen aus drei Steinen mit einer Lücke einen geringeren Wert erhalten, als das beschriebene Dreieck. Reihen, die eine Angriffsfläche bieten, tragen vergleichsweise wenig zum Bewertungswert bei.

Muster						
Bewertung	5	5	20	41	$-\infty$	∞

Tabelle 2.1: Bewertungen der einzelnen Muster durch die Bewertungsfunktion für Alpha-Beta

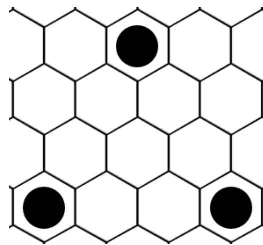


Abbildung 2.4: Vorteilhaftes Dreiecksmuster

2.2.2 Iterative Tiefensuche

Die iterative Tiefensuche (engl. iterative deepening depth-first search) bildet eine Kombination aus Tiefen- und Breitensuche. Es wird eine Alpha-Beta-Suche mit steigender Tiefe ausgeführt. Beginnend mit einer Suchtiefe von eins, wird die Suche mit den folgenden Tiefen (2,3 usw.) wiederholt, bis eine maximale Tiefe erreicht ist oder eine zeitliche Beschränkung überschritten wird [11]. Im Kontext dieser Arbeit wird eine zeitliche Beschränkung verwendet, um Vergleichbarkeit mit den Monte Carlo Tree Search Ansätzen zu schaffen, welche ebenfalls eine solche Beschränkung verwenden. Der Vorteil dieses Verfahrens besteht darin, dass die Ergebnisse der bisher untersuchten Suchtiefen die Zugsortierung verbessern können, indem der als bisher bestbewertete Zug zu erst untersucht wird.

Außerdem können Siege in niedrigen Suchtiefen schneller gefunden werden, wie bei einer Breitensuche.

Nach Korf [11] minimiert die iterative Tiefensuche Laufzeit und Speicherplatzbedarf asymptotisch für eine gegebene Suchtiefe. Daraus folgt, dass für eine gebene Laufzeit die erreichte Suchtiefe asymptotisch maximiert wird. Unter der Annahme, dass die Qualität der Bewertung bei der Alpha-Beta-Suche mit zunehmender Suchtiefe steigt, ist dies vorteilhaft.

Im Falle des Ablaufens der zur Verfügung stehenden Zeit kann der Algorithmus abgebrochen werden und das Ergebnis der letzten vollständig berechneten Tiefe zurückgegeben werden. So wird die maximale Suchtiefe, welche innerhalb der zeitlichen Beschränkung berechenbar ist, erreicht.

2.2.3 Transpositionstabelle

Während der Alpha-Beta-Suche ist es möglich, dass eine Spielsituation mehrmals untersucht wird, da eine Änderung in der Reihenfolge von einer Menge an Zügen zu unterschiedlichen Knoten im Suchbaum führt, aber die zugrundeliegende Spielsituation die gleiche ist. Da die Bewertung einer Spielsituation unabhängig vom Pfade, der zu ihr führt, ist, werden dann redundante Berechnungen durchgeführt. Um dies zu vermeiden wird eine Transpositiontabelle verwendet. Bereits besuchte Knoten bekommen einen Eintrag in dieser Hashtabelle, wobei folgendes gespeichert wird [1]:

- **hash**: Aus der aktuellen Spielstellung wird ein Hashwert mittels Zobrist Hashing generiert. Dieser wird als Index für die Tabelle verwendet.
- **move**: Der beste Zug aus der aktuellen Spielstellung.
- **score**: Die Bewertung des besten Zuges. **score** kann ein genauer Wert, eine Ober- oder eine Untergrenze sein.
- **flag**: Kennzeichnet die Art der Bewertung **score** [15] [14]:
  - *EXACT*: Genauer Wert. Falls $\alpha <$ **score** $< \beta$.
  - *LOWERBOUND*: Untergrenze. Falls $\beta \leq$ **score**.
  - *UPPERBOUND*: Obergrenze. Falls **score** $\leq \alpha$.
- **depth**: Die relative Tiefe im Suchbaum. Wenn n die Suchtiefe der Alpha-Beta-Suche ist und bereits m Züge ausgewählt wurden, ist die relative Tiefe $n - m$.

Bevor ein Knoten untersucht wird, schaut der Algorithmus nach, ob die Spielposition bereits in der Transpositiontablle vorhanden ist. Falls die Tiefe des noch zu untersuchenden Teilbaums kleiner oder gleich der gespeicherten Tiefe **depth** ist, hängt das Verhalten des Algorithmus von **flag** ab [1]. Gilt **flag** = *EXACT*, so muss der Knoten nicht weiter untersucht werden und die gespeicherte Bewertung wird zurückgegeben. Andernfalls wird α (**flag** = *LOWERBOUND*), bzw. β (**flag** = *UPPERBOUND*), mit dem gespeicherten Wert aktualisert. Dabei wird α auf das Maximum von α und **score** gesetzt und β analog auf das Minimum. Falls dann $\beta \leq \alpha$ gilt, kann ein alpha-beta cutoff ausgeführt werden. Wurde kein cutoff vorgenommen oder ist die Tiefe des noch zu untersuchenden Teilbaums größer als

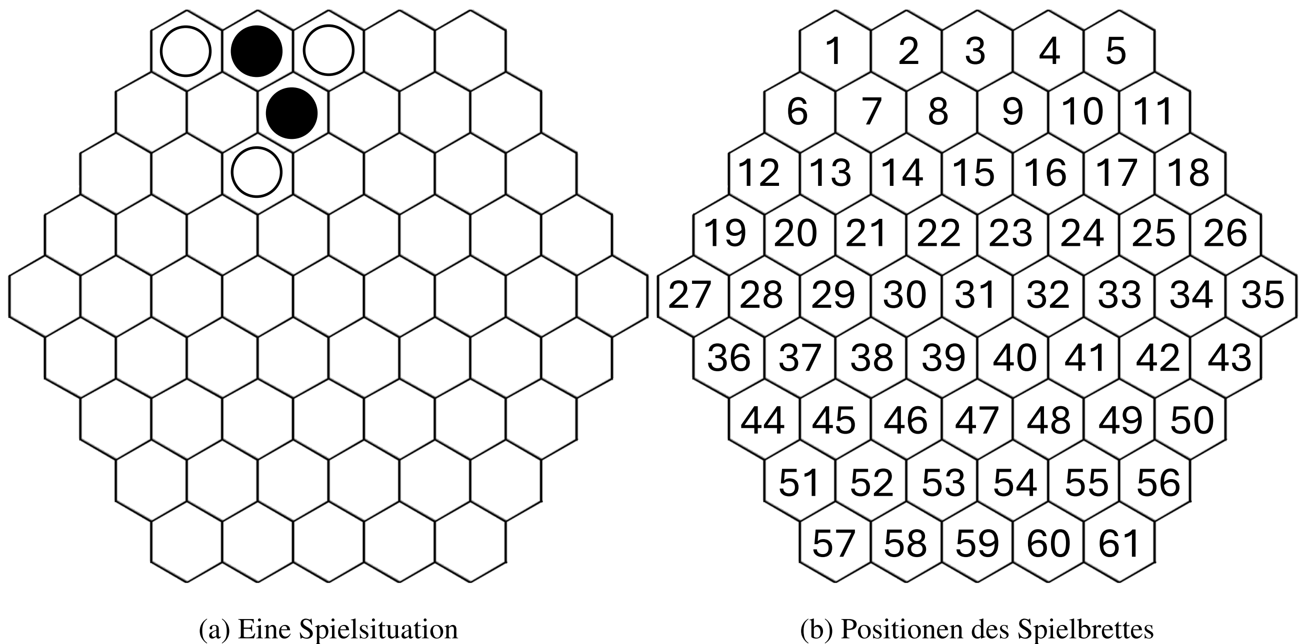
depth, wird der gespeicherte Zug genutzt, um die Zugsortierung des Algorithmus zu verbessern und wird als Erstes untersucht.

Zobrist Hashing

Vor der Suche wird eine Menge an Pseudozufallszahlen generiert [16]. Im Fall von Yavalath sind es $122 = 2 \cdot 61$, denn es wird für beide Spieler je eine Zahl für jede Position auf dem Spielbrett verwendet. Um einen Hash aus einer Spielsituation zu generieren werden für beide Spieler die Pseudozufallszahlen der Positionen an denen sie Steine platziert haben, mittels XOR miteinander verknüpft. XOR ist der bitweise exklusive ODER-Operator, auch geschrieben als $\oplus$.
Es werden also Pseudozufallszahlen $s_{i,j}$ generiert, wobei $i \in [1, 61]$ die Position auf dem Spielbrett und $j \in \{1, 2\}$ der Spieler ist. Der Hashwert des Brettes in Abbildung 2.5a wird dann berechnet als:

$$\text{hash} = s_{1,1} \oplus s_{3,1} \oplus s_{14,1} \oplus s_{2,2} \oplus s_{8,2}$$

Die Größe der Pseudozufallszahlen und somit auch der Hashwerte haben Auswirkungen auf die Qualität des Verfahrens. Sind sie zu klein so steigt die Gefahr einer Kollision der Hashwerten von verschiedenen Spielsituationen. Große Zahlen hingegen sorgen für einen größeren Rechenaufwand. Um eine Balance zwischen diesen potenziellen Problemen zu treffen werden in dieser Arbeit 64-bit unsigned Integer verwendet.



(a) Eine Spielsituation

(b) Positionen des Spielbrettes

Abbildung 2.5: Beispiel für die Berechnung von Zobrist Hashwerten

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) ist eine Technik, um Züge mittels zufälliger Simulationen und dem Aufbauen eines Suchbaumes anhand der Ergebnisse, zu finden [4]. MCTS ist ein anytime Algorithmus, also sind die durch den Algorithmus gefunden Züge nicht zwangsweise optimal. Aber mit

steigender Rechenleistung bzw. -zeit nähert sich das Ergebnis dem Optimum an. MCTS baut innerhalb eines Rechenbudgets iterativ einen Suchbaum auf. Als Rechenbudget dient meist eine zeitliche Beschränkung. Eine gegebene Anzahl an Iterationen wäre eine weitere mögliche Einschränkung. Die Knoten repräsentieren eine Spielstellung, von der aus die Spielstellung der Kinderknoten mit einem der möglichen Spielzüge erreicht werden. Im Weiteren wird die Funktion f verwendet, sodass $f(s, a)$ die Spielstellung ist, welche durch das Ausführen der Aktion a in Spielstellung s entsteht.

Für einen Knoten v werden folgende Informationen gespeichert:

- $s(v)$: die Spielstellung des Knotens.
- $N(v)$: wie oft der Knoten besucht wurde.
- $Q(v)$: der Reward des Knotens; wird mit 0 initialisiert.
- $p(v)$: der Parentknoten von v.
- $a(v)$: die Aktion, sodass $s(v) = f(s(p(v)), a)$.

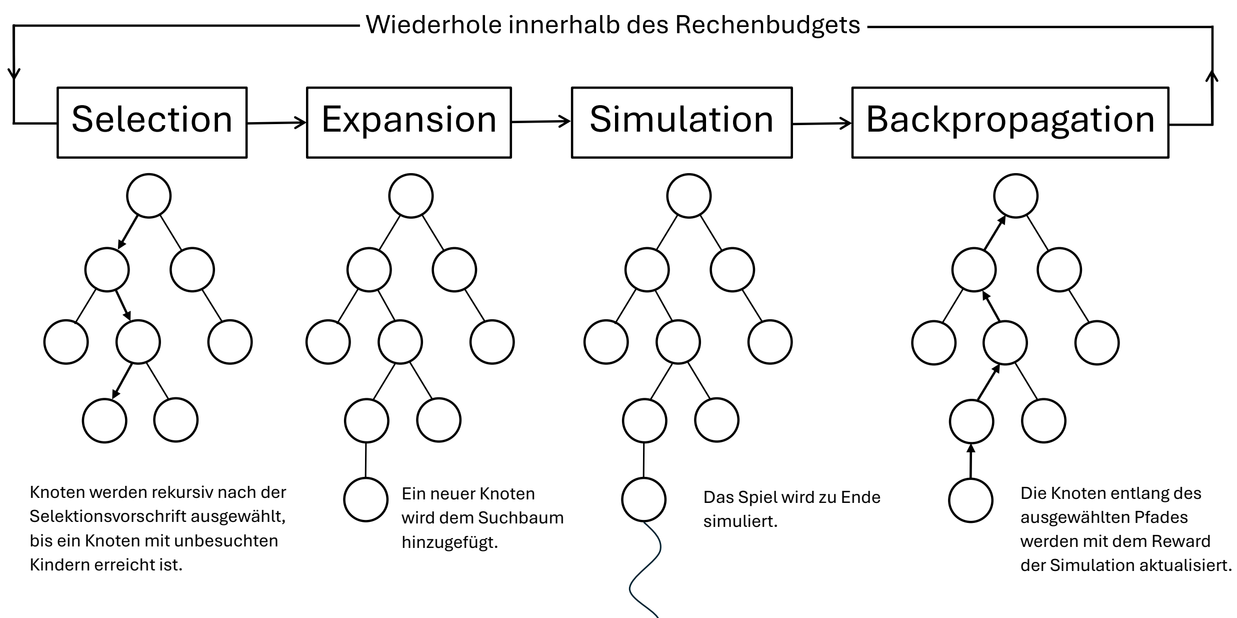


Abbildung 2.6: Eine Iteration des Monte Carlo Tree Search. Nach [5]

In jeder Iteration des Algorithmus werden die folgenden vier Schritte ausgeführt [5]:

Selection

Der bereits aufgebaute Suchbaum wird mit einer Selektionsvorschrift durchlaufen, bis ein Knoten erreicht wird, welcher noch nicht vollständig erweitert ist und keine Endstellung repräsentiert. Ein Knoten ist nicht vollständig erweitert, wenn es mögliche Kinderknoten gibt, die noch nicht Teil des Suchbaums sind oder noch nicht besucht wurden [4]. Die Selektionsvorschrift wählt das beste Kind

eines Knotens so aus, dass sowohl bisher wenig besuchte Knoten (Exploration), als auch Knoten mit guten Ergebnissen (Exploitation) ausgewählt werden können. Wie das beste Kind bestimmt wird, wird weiter unten in dem Abschnitt *Upper Confidence Bounds angewandt auf Bäume (UCT)* erläutert.

Expansion

Ein bisher unbesuchtes Kind des Knotens wird generiert und dem Baum hinzugefügt.

Algorithmus 3 Expansionschritt nach [4].

1: **function** EXPANSION(v)
2: wähle a aus bisher unausgewählten Aktionen in v.
3: Füge neues Kind v' von v zum Suchbaum hinzu, mit $s(v') = f(s(v), a)$ und $a(v') = a$
4: **return** v'
5: **end function**

Simulation

Von der Spielstellung des neu generierten Knotens aus, wird das Spiel zu Ende simuliert. Dabei ist es üblich die Züge zufällig zu wählen, um die Rechenkapazitäten, die für die Simulation benötigt werden, zu minimieren. Es kann allerdings auch sinnvoll sein, eine andere Strategie zu wählen, welche die Zeit, aber auch die Qualität, der Simulation steigert. Eine Simulation für Yavalath, in der ein Spieler beispielsweise eine Reihe aus drei Steinen bildet ohne dazu gezwungen zu sein und dadurch verliert, ist keine repräsantative Simulation, um die Qualität der möglichen Züge zu bewerten.

Algorithmus 4 Simulationschritt mit zufälliger Zugwahl nach [4]

1: **function** SIMULATION(s)
2: **while** s ist kein Endstellung **do**
3: Wähle $a \in A(s)$ gleichverteilt zufällig.
4: $s \leftarrow f(s, a)$
5: **end while**
6: **return** Reward für Spielstellung s ▷ Reward ist das Ergebnis des Spiels (Algorithmus 1)
7: **end function**

Backpropagation

In diesem Schritt werden die Knoten entlang des in dieser Iteration gewählten Weges durch den Baum mit dem Ergebnis der Simulation aktualisert. Da Yavalath von zwei Spielern gespielt wird und diese abwechselnd Züge wählen, ist der Reward für den Gewinner positiv und für den Verlierer negativ.

Algorithmus 5 Backpropagationschritt nach [4]

```
1: function BACKPROPAGATION(v, Δ)
2:     while v is not null do
3:         N(v) ← N(v) + 1
4:         Q(v) ← Q(v) + Δ
5:         Δ ← −Δ
6:         v ← p(v)
7:     end while
8: end function
```

Diese vier Schritte werden dann, wie in Algorithmus 6 beschrieben, wiederholt ausgeführt. Wie bereits erwähnt, wird die Implementation von BESTCHILD in der nächsten Sektion beschrieben.

Algorithmus 6 Monte Carlo Tree Search nach [4]

```
1: function MCTS(s_0)
2:     v_0 ← neuer Knoten mit Spielstellung s_0        ▷ v_o ist der Wurzelknoten des Suchbaums
3:     while innerhalb Rechenbudget do
4:         v ← SELECTION(v_0)
5:         Δ ← SIMULATION(s(v))
6:         BACKPROPAGATION(v, Δ)
7:     end while
8:     return a(BESTCHILD(v_0, 0))
9: end function
```

2.3.1 Upper Confidence Bounds angewandt auf Bäume (UCT)

Die Art und Weise, wie der Suchbaum generiert wird, ist stark Abhängig von der Selektionsvorschrift. Es bietet sich an die Wahl des Kinderknotens beim traversieren des Baumes als Multi-Armed Bandit Problem zu betrachten, wie von Kocsis und Szepesvári in 2006 vorgeschlagen [10]. Dafür wird der Ausdruck

$$UCT = \overline{X_j} + C_p \sqrt{\frac{\ln N}{n_j}} \tag{2.1}$$

maximiert, wobei $\overline{X_j}$ der durchschnittlichen Rewards der Kinder j, $C_P > 0$ eine Konstante, N die Anzahl der Besuche des aktuellen Knotens und n_j die Anzahl der Besuche der Kinderknoten sind. Der Ausdruck $\overline{X_j}$ sorgt dafür, dass gut bewertete Knoten bevorzugt werden (Exploitation). Durch den zweiten Teil des Terms wird sicher gestellt, dass trozdem alle Kinder ausgewählt werden können (Exploration). Dadurch, dass der Nenner n_j durch eine steigende Anzahl an Besuchen wächst, steigt der UCT-Wert von bisher weniger ausgewählte Knoten durch die Exploration [4].

Insbesondere wird BESTCHILD am Ende des MCTS mit $c = 0$ aufgerufen (Algorithmus 6 in Zeile 8), da dort der durch den Algorithmus bestbewertete Zug zurückgegeben wird und darauf die Explotation keinen Einfluss mehr haben darf.

Algorithmus 7 Wahl der Kinder beim traversiern des Baums durch UCT nach [4]

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\operatorname{argmax}} \frac{Q(v')}{N(v')} + c\sqrt{\frac{\ln N(v)}{N(v')}}$
end function

2.3.2 Simulation mit General Domain Knowledge

Um die Qualität der Simulationen zu steigern, ist es sinnvoll die Züge nicht komplett zufällig zu wählen. Stattdessen schlägt Browne [2] vor die Simulationen durch allgemeines Wissen über die Domäne von Brettspielen (engl. General Domain Knowledge) zu verbessern. Konkret bedeutet dies, dass gewinnende Züge priorisert werden. An zweiter Stelle stehen Züge, die einen Sieg des Gegner im nächsten Zug blockiert. Falls solche Züge nicht existieren wird ein Zug gespielt, der das Spiel nicht sofort verliert (durch eine Reihe aus 3 Steinen). Der folgende If-Block ersetzt dann Zeile 3 in SIMULATION (Algorithmus 4).

if mindestens ein gewinnender Zug existiert **then**
  Wähle a gleichverteilt zufällig von diesen
else if mindestens ein blockierender Zug exisiert (der nicht verliert) **then**
  Wähle a gleichverteilt zufällig von diesen
else if mindestens ein nicht verlierender Zug existiert **then**
  Wähle a gleichverteilt zufällig von diesen
else
  Wähle $a \in A(s)$ gleichverteilt zufällig
end if

2.3.3 Eigenschaften von MCTS

Im Gegnsatz zur Alpha-Beta-Suche benötigt Monte Carlo Tree Search keine heuristische Bewertungsfunktion und kommt somit ohne spezifisches Wissen über den Anwendungsbereich aus. Dadruch kann es ein starker Algorithmus zum Spielen von unbekannten, wenig erforschten Spielen sein. Im Fall von taktischen Spielen wie Yavalath kann es allerdings zu plötzlichen Spielenden kommen, wobei viele Züge nicht substanziel zum Spielziel beitragen [6]. Auf Grund dessen ist es möglich, dass MCTS oft eher schwache Züge findet, da sich die Bewertung der Knoten nicht schnell genug an den tatsächlichen Wert des Zuges annähern. Um dem engegenzusteuern gibt es verschiedene Ansätze andere Algorithmen mit MCTS in Kombination einzusetzten.

2.3.4 MCTS in Kombination mit Alpha-Beta-Suche

level-k sudden death/win

Eine Spielsituation befindet sich in einem level-k sudden death, falls der Spieler einen Zug von einer Menge an bestimmten Zügen spielen muss, damit sein Gegner nicht einen Sieg, innerhalb von k Zügen, für sich erzwingen kann. Analog ist ein level-k sudden win eine Situation in der der Spieler innerhalb von k Zügen einen Sieg forcieren kann [12]. Da sich die Spieler abwechseln ist k für ein

sudden death immer gerade und bei einem sudden win ungerade. Alpha-Beta erkennt diese Situationen durch die Bewertung des Wurzelknotens automatisch, da dieser einen garantierten Sieg oder eine garantierte Niederlage anzeigen kann.

Integration von Alpha-Beta in MCTS

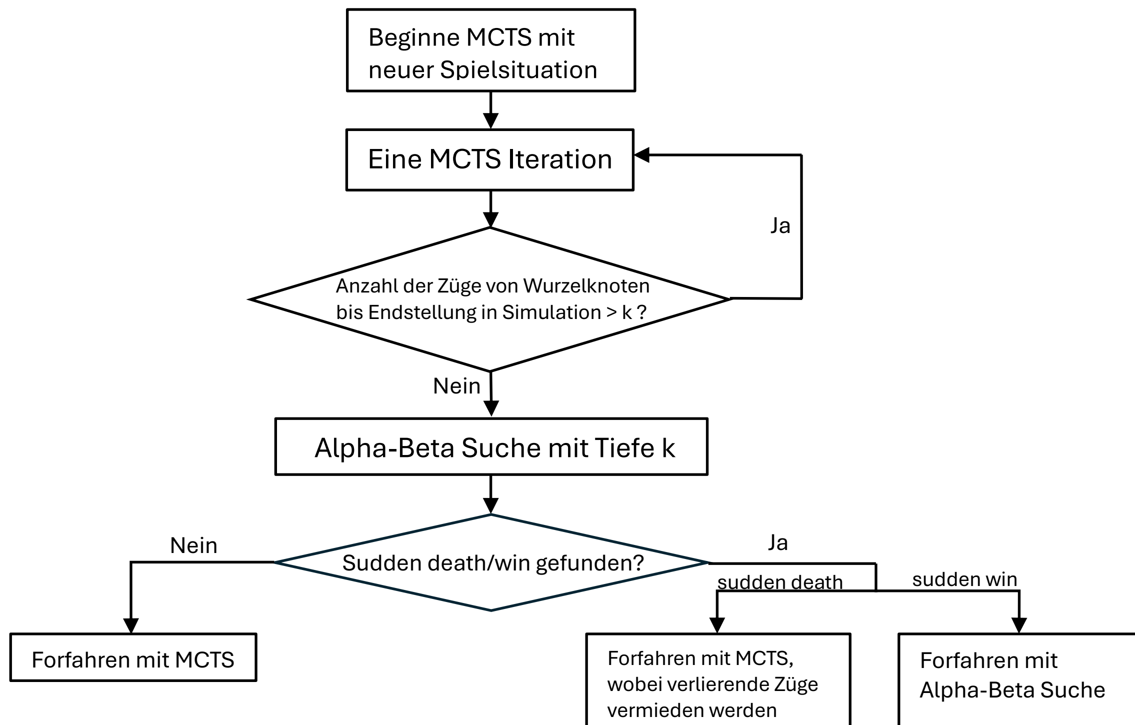


Abbildung 2.7: Flussdiagram von MCTS mit Alpha-Beta nach [12]

Um das Rechenbudget für MCTS nicht unnötig durch die Ausführung der Alpha-Beta-Suche zu verringern, schlägt Li [12] vor dies nur zu tun falls es nötig ist:
Nach jeder Iteration wird überprüft, ob die Anzahl der Züge vom Wurzelknoten bis zur Endstellung der Simulation gößer als k ist. Ist dies der Fall so wird diese Überprüfung nach der nächsten Iteration wiederholt. Andernfalls wird eine Alpha-Beta-Suche ausgeführt. Wird ein sudden win gefunden, so wird dieser mit der Alpha-Beta-Suche ausgespielt. Andernfalls wird mit MCTS fortgefahren, wobei im Fall eines gefundenen sudden death die Züge, welche zu diesem führen, vermieden werden. Mit dieser Vorgehensweise ist es nicht garantiert eine sudden death / win Situation immer zu erkennen. Für das Spiel Gomoku ist die Wahrscheinlichkeit nahe 1 für eine größe Anzahl an MCTS-Iterationen, eine moderate Größe des Spielbrettes und einer kleinen Zahl k [12]. Es liegt nahe, dass dies für Yavalath ähnlich ist.

2.3.5 MCTS in Kombination mit Poof Number Search

Proof Number Search (PNS)

Die Proof Number Search ist eine Methode um zu zeigen[3], dass in einer gegebenen Spielstellung ein Sieg erzwungen werden kann. Der Suchbaum gilt als bewiesen, wenn ein Sieg erzwungen werden kann, bzw. als widerlegt, wenn sich eine Niederlage oder ein Unentschieden nicht verhindern lässt, und als unbekannt wenn noch nicht gezeigt werden konnte, dass einer der Spieler das Ende des Spiels forcieren kann [6]. Die Knoten des Baumes sind ODER- oder UND-Knoten, welche sich zwischen den einzelnen Tiefen im Suchbaum abwechseln. Des Weiteren hat jeder Knoten eine proof number (pn) und eine disproof number (dpn). Die proof number eines Knotens gibt die Anzahl an Blattknoten an, welche noch mindestens bewiesen werden müssen, um den Knoten zu beweisen. Die disproof number gibt dies analog für das Widerlegen des Knotens an.

Abbildung 2.8: PNS Suchbaum mit proof und disproof numbers nach [6]

Da es das Ziel ist einen garantierten Sieg zu beweisen, gelten Blattknoten bei einem Sieg als bewiesen und haben pn = 0 und dpn = ∞. Bei einer Niederlage oder Unentschieden gilt für Blattknoten pn = ∞ und dpn = 0. Ist das Spiel an einem Blattknoten noch nicht beendet ist sowohl pn = 1 als auch dpn = 1. Die Werte der inneren Knoten ergeben sich aus den Werten ihrer Kinder. Die proof number eines inneren ODER-Knotens ist das Minimum der pn seiner Kinder, da es um den Knoten zu beweisen ausreicht dies für eines der Kinder zu tun. Für innere UND-Knoten ist die proof number gleich der Summe der pn aller Kinder des Knotens, da alle Kinder bewiesen werden müssen, um den Knoten selbst zu beweisen. Für die disproof number ist dies genau umgekehrt. Das heißt die dpn von ODER-Knoten ist gleich der Summe der dpn seiner Kinder und von UND-Knoten gleich dem Minimum der dpn seiner Kinder.

[3] oder zu widerlegen

Wie bei MCTS auch wird in jeder Iteration ein Blattknoten ausgewählt, welcher dann erweitert wird indem seine Kinder generiert und ausgewertet werden. Die Wahl dieses Knoten, auch most-proving node genannt, beginnt bei der Wurzel. Von dort werden bei ODER-Knoten das Kind mit der kleinsten proof number und bei UND-Knoten mit der kleinsten disproof number ausgewählt, bis ein Balattknoten erreicht ist. Bei Gleichständen wird zufällig gewählt. In Abbildung 2.8 ist der Weg durch den Baum für die Wahl der most-proving node dick markiert.

Proof Number MCTS (MCTS-PN)

Proof Number Search wird durch eine Modifikation der UCT Formel in den MCTS Algorithmus integriert. Statt die Werte der (dis)proof number direkt zu benutzen, dienen sie dazu eine Rangfolge der Knoten zu bestimmen [6]. Für ODER-Knoten erhält das Kind mit der kleinsten proof number den besten Rang. Bei UND-Knoten hat das Kind mit der kleinsten disproof number den besten Rang. In beiden Fällen ist dann das Kind mit der zweitkleinsten (dis)proof number Rang 2 und so fort. Haben zwei Kinderknoten die gleichen Werte so teilen sie sich einen Rang. Die modifizierte Formel lautet wie folgt:

$$UCT\text{-}PN = \overline{X_j} + C_p\sqrt{\frac{\ln n}{n_j}} + C_{pn}\left(1 - \frac{\mathrm{pnRank}_j}{\mathrm{argmax}_{i\in\mathrm{children}}(\mathrm{pnRank}_i)}\right) \tag{2.2}$$

wobei pnRank_j der Rang des Knotens j ist und $\mathrm{argmax}_{i\in\mathrm{children}}(\mathrm{pnRank}_i)$ der höchste Rang aller Kinder ist. Damit wird der Rang also auf das Interval $[0, 1]$ normalisiert. C_{pn} ist der neue Parameter, welcher den Einfluss des neuen Teils der Formel beeinflusst. Der Rest der Formel bleibt wie oben beschrieben.

Um MCTS zu PN-MCTS zu modifizieren bedarf es, neben der Anpassung von BESTCHILD mit der UCT-PN Formel, lediglich der Aktualisierung der (d)pn Werte aller ausgewählter Knoten während des Backpropagationschrittes.

3 Ergebnisse

Die Tests wurden auf einem System mit dem CPU AMD Ryzen 5 5600X (3700 Mhz) und 16 GB RAM ausgeführt. Das Betriebssystem war Windows 10 und für die Implementierung der Algorithmen wurde Python 3.11.9 mit der Bibliothek NumPy verwendet. Zur Verarbeitung der Erhobenen Daten und Visualisierung der Ergebnisse wurden außerdem die Bibliotheken Matplotlib, Pandas und Seaborn benutzt.

3.1 Berechnungszeit Alpha-Beta-Suche

Die untersuchen Varianten des Algorithmus wurden ohne zeitliche Begrenzung in 10 verschiedenen Spielstellungen, mit steigender Anzahl bereits gespielter Züge, mit drei Suchtiefen ausgeführt. Die verwendeten Spielstellungen sind in Abbildung 3.11 zu sehen. Die untersuchen Algorithmen sind:

- **AB**: Alpha-Beta ohne Verbesserungen
- **AB-TT**: Alpha-Beta mit Transpositionstabelle
- **AB-Iter**: Alpha-Beta mit iterativer Tiefensuche
- **AB***: Alpha-Beta mit iterativer Tiefensuche und Transpositionstabelle

In Tabelle 3.1 sind die durchschnittlichen Berechnungszeiten jedes Algorithmus für die Suchtiefen angegeben. Da die Alpha-Beta-Suche deterministisch ist wurden alle Varianten nur einmal auf jede Spielsituation angewendet. Aus diesen Werten wurden für Abb. 3.1 relative Rechenzeiten berechnet, wobei die größte Zeit einer Suchtiefe als Referenzwert für die anderen Zeiten dieser Tiefe dient. Somit wurden alle Werte einer Tiefe relativ zur jeweils langsamsten Laufzeit skaliert, ein direkter Vergleich zwischen Suchtiefen ist in dieser Abbildung also nicht möglich. Außerdem sind die einzelnen Laufzeiten der Algorithmen für jede in Abbildung 3.2 zu sehen.

Suchtiefe	**AB**	**AB-TT**	**AB-Iter**	**AB***
4	1.61	1.53	0.81	0.43
5	22.25	17.99	13.27	4.89
6	259.08	156.87	100.44	23.54

Tabelle 3.1: Durchschnittliche Berechnungszeit der Alpha-Beta Varianten je Suchtiefe (in Sekunden)

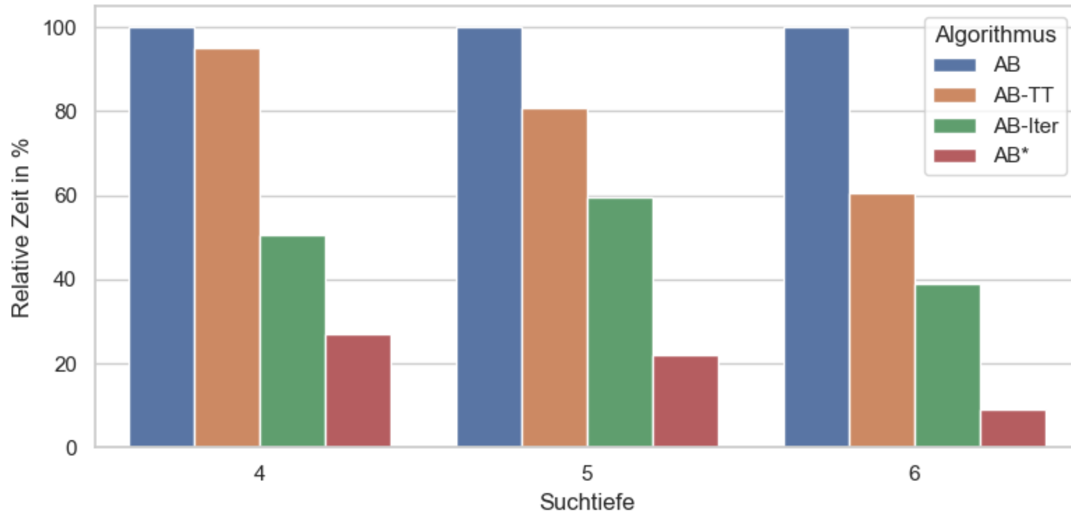
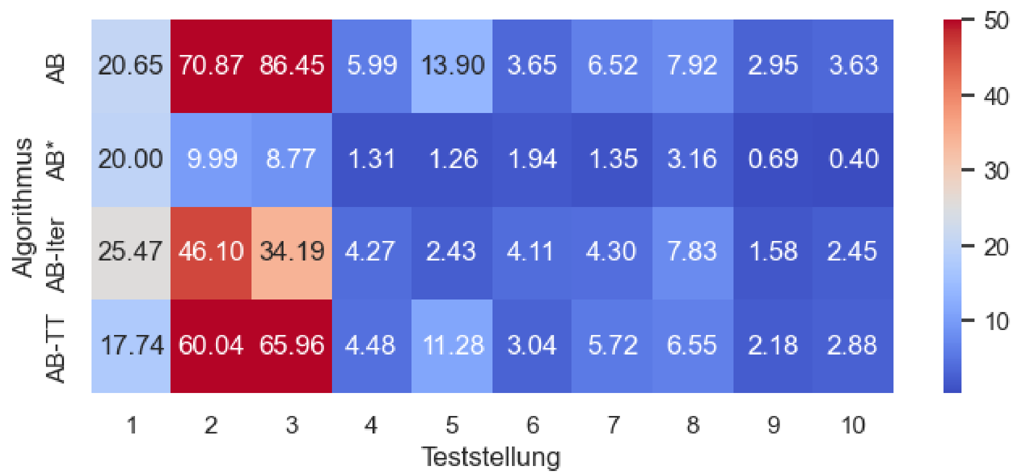
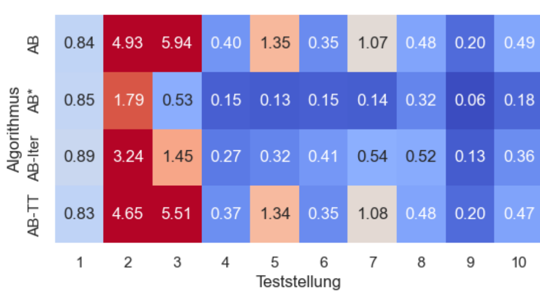


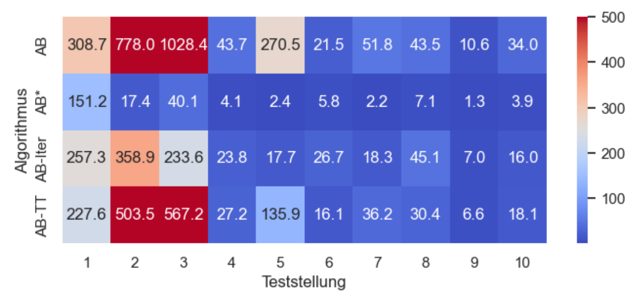
Abbildung 3.1: Relative durchschnittliche Berechnungszeit der Alpha-Beta Varianten pro Suchtiefe und Algorithmus



(a) Suchtiefe 5



(b) Suchtiefe 4



(c) Suchtiefe 6

Abbildung 3.2: Laufzeiten der Alpha-Beta Varianten pro Spielposition (in Sekunden)

3.2 Wahl der MCTS Konfiguration

Im Weiteren werden folgende Bezeichnungen verwendet:

- **MCTS**: Die Basis Variante von MCTS, ohne die Hilfe von Alpha-Beta oder Proof Number Search
- **MCTS-AB**: MCTS mit Alpha-Beta zur Erkennung von sudden death/win Situationen
- **MCTS-PN**: MCTS mit der modifizierten UCT-PN Formel

Für Tests in denen verschiedene c-Werte verwendet werden, sind diese mit angegeben. Die Verwendung von GDK wird ebenfalls deklariert. Zur Bestimmung einer geeigneten Konfiguration wurden **MCTS** mit verschiedenen Parametern im direkten Vergleich in mehreren Testschritten ausgeführt. In jedem Schritt wurden in jedem Matchup insgesamt 60 Spiele simuliert, wobei jeder Algorithmus in je der Hälfte der Spiele als Startspieler agierte. Alle Abbildungen in diesem Abschnitt, welche durchschnittliche Siegesraten zeigen, enthalten Fehlerbalken, welche den Standardfehler der entsprechenden Mittelwerte anzeigen.

1. Testschritt

Das Bestimmen einer geeigneten Konfiguration des Algorithmus startete durch den direkten Vergleich von **MCTS** mit unterschiedlichen Explorationskonstanten auf einem leeren Spielbrett, um die Fairness zwischen den schwarzen und weißen Steinen zu garantieren.

In einem ersten Testdurchlauf wurden $c \in \{0.5, 1, \sqrt{2}, 2\}$ gewählt, da diese ein weites Spektrum an Werten umfassen. Als Zugzeit wurde 1 Sekunde gewählt.

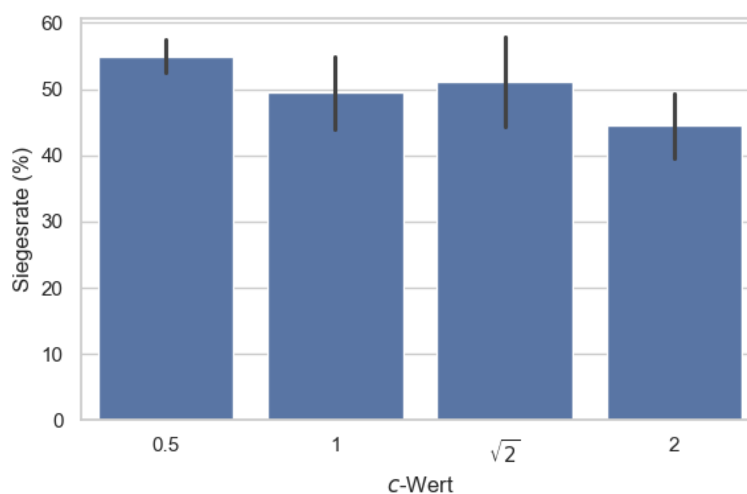


Abbildung 3.3: Testschritt 1: Durchschnittliche Siegesrate der c-Werte im Spiel untereinander ohne GDK (Zugzeit 1 Sekunde)

2. Testschritt

Da die beiden Werte mit den meisten Siegen (0.5 und $\sqrt{2}$) im direkten Vergleich je die Hälfte der Spiele gewannen, und die geringe Zeit pro Zug die Leistung der Werte verschieden beeinflussen könnte, wurde das Experiment für diese Werte mit einem größeren Zeitbudget von 10 Sekunden wiederholt.

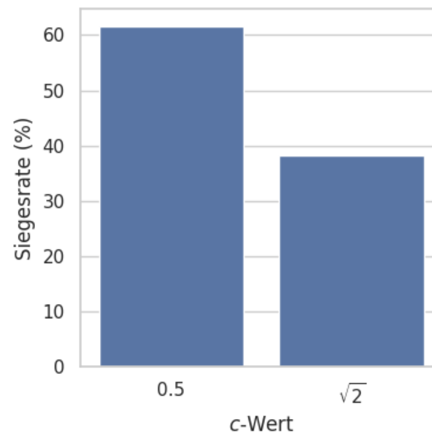


Abbildung 3.4: Testschritt 2: Siegesrate der weiter untersuchten c-Werte (Zugzeit 10 Sekunden)

3. Testschritt

Mit der bestimmten Konstante $c = 0.5$ wurden **MCTS** mit und ohne Simulationen mit General Domain Knowledge gegeneinander mit einem Zeitbudget von 10 Sekunden pro Zug getestet. Hier wurde ebenfalls eine Anzahl von 60 Partien, mit je abwechselnden Startspielern, gespielt.

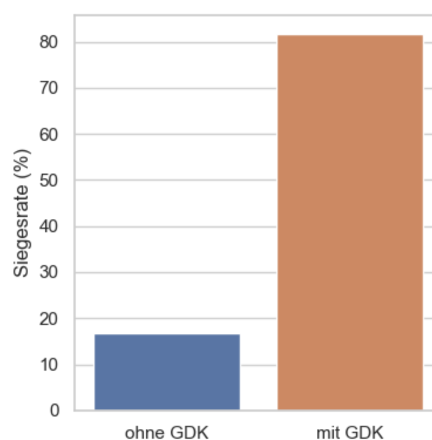


Abbildung 3.5: Testschritt 3: Siegesraten für **MCTS** ohne GDK gegen **MCTS** mit GDK und jeweils $c = 0.5$ (Zugzeit 10 Sekunden)

4. Testschritt

Um zu Überprüfen, welchen Einfluss die Verwendung von General Domain Knowledge auf die Leistung des Algorithmus mit verschiedenen c-Werten hat, wurden die Experimente ähnlich wie in Testschritt 1. mit einer Zugzeit von 1 Sekunde wiederholt. Diesmal wurde die Beobachtung aus den vorherigen Durchläufen berücksichtigt, dass geringere Explorationswerte den Algorithmus verstärken zu scheinen, und mit den Explorationskonstanten $c \in \{0.3, 0.5, 0.7, \sqrt{2}\}$ getestet.

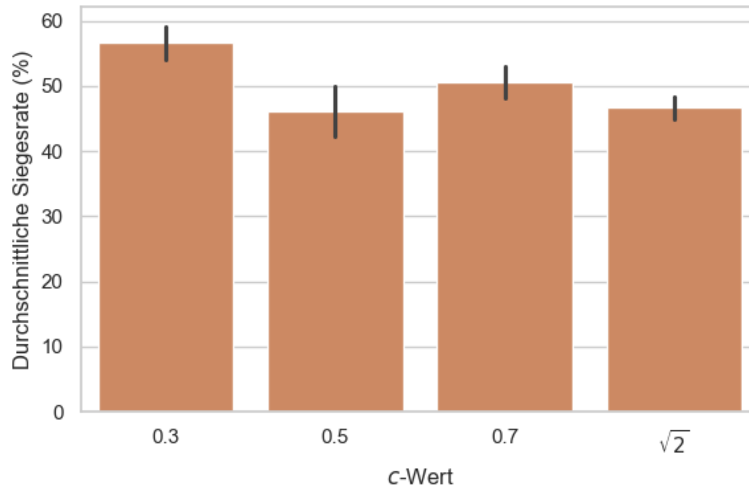


Abbildung 3.6: Testschritt 4: Durchschnittliche Siegesrate der c-Werte im Spiel untereinander mit GDK (Zugzeit 1 Sekunde)

Kandidat	Gegner			
c-Wert	0.3	0.5	0.7	$\sqrt{2}$
0.3	–	62	53	55
0.5	38	–	50	50
0.7	47	50	–	55
$\sqrt{2}$	45	50	45	–

Tabelle 3.2: Testschritt 4: Siegesrate (in %, gerundet) der untersuchten c-Werte für **MCTS** mit GDK (Zugzeit 1 Sekunde). Unentschieden werden als nicht-Sieg für beide gewertet.

5. Testschritt

Abschließend wurde das Verfahren wieder mit einer höheren Berechnungszeit von 5 Sekunden pro Zug wiederholt, um den Einfluss der Berechnungszeit genauer zu untersuchen, aus Gründen der zeitlichen Limitationen wurde dabei $c = 0.5$ nicht mehr betrachtet, da dieser für 1 Sekunde die schlechteste Siegesrate verzeichnete.

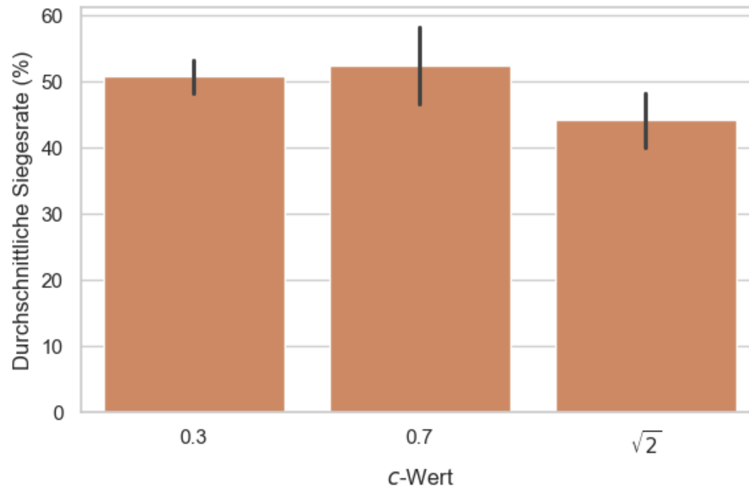


Abbildung 3.7: Testschritt 5: Durchschnittliche Siegesrate der c-Werte im Spiel untereinander mit GDK (Zugzeit 5 Sekunden)

Kandidat	Gegner		
c-Wert	0.3	0.7	$\sqrt{2}$
0.3	–	53	48
0.7	47	–	58
$\sqrt{2}$	48	40	–

Tabelle 3.3: Testschritt 5: Siegesrate (in %, gerundet) der untersuchten c-Werte für **MCTS** mit GDK (Zugzeit 5 Sekunden). Unentschieden werden als nicht-Sieg für beide gewertet.

3.3 Vergleich der ausgewählten Algorithmen

Basierend auf den vorherigen Experimenten wurden die Konfigurationen der Algorithmen wie folgt ausgewählt:

AB*: Alpha-Beta-Suche mit iterativer Tiefensuche und Transpositionstabelle

MCTS, MCTS-AB, MCTS-PNS mit General Domain Knowledge und $c = 0.7$

Für die Konstante C_{pn} des **MCTS-PN** ist 1 gewählt worden, da nach Doe et al. [6] sich dieser Wert in anderen Anwendungsbereichen als vorteilhaft herausgestellt hat.

Die Algorithmen traten zwei mal im Round-Robin-Verfahren gegeneinander an. Alle Algorithmen erhielten das gleiche Zeitbudget. Für ersten Durchlauf betrug die Zugzeit 10 Sekunden und es wurde mit einem leeren Brett begonnen. Während des zweiten Durchlaufes hatten die Algorithmen nur 5 Sekunden pro Zug und das Spiel startete in Tesstellung 4 (Abb. 3.11d). Diese Teststellung wurde gewählt, da die von Browne zur Verfügung gestellte Implementierung eines MCTS-Yavalath-Spieler [1] in dieser Stellung einen Zug mit einer Siegchance von ungefähr 50% findet.

[1]Zu finden unter: https://cambolbro.com/games/yavalath/ (Zugriff am 27.5.2025)

3.3.1 Anzahl der Iterationen

Für die drei MCTS Algorithmen wurden für jedes Spiel die durchschnittliche Anzahl an durchgeführten Iterationen festgehalten. Jeder Datenpunk entspricht dem Durchschnitt der Iterationen während eines einzelnen Spiels.

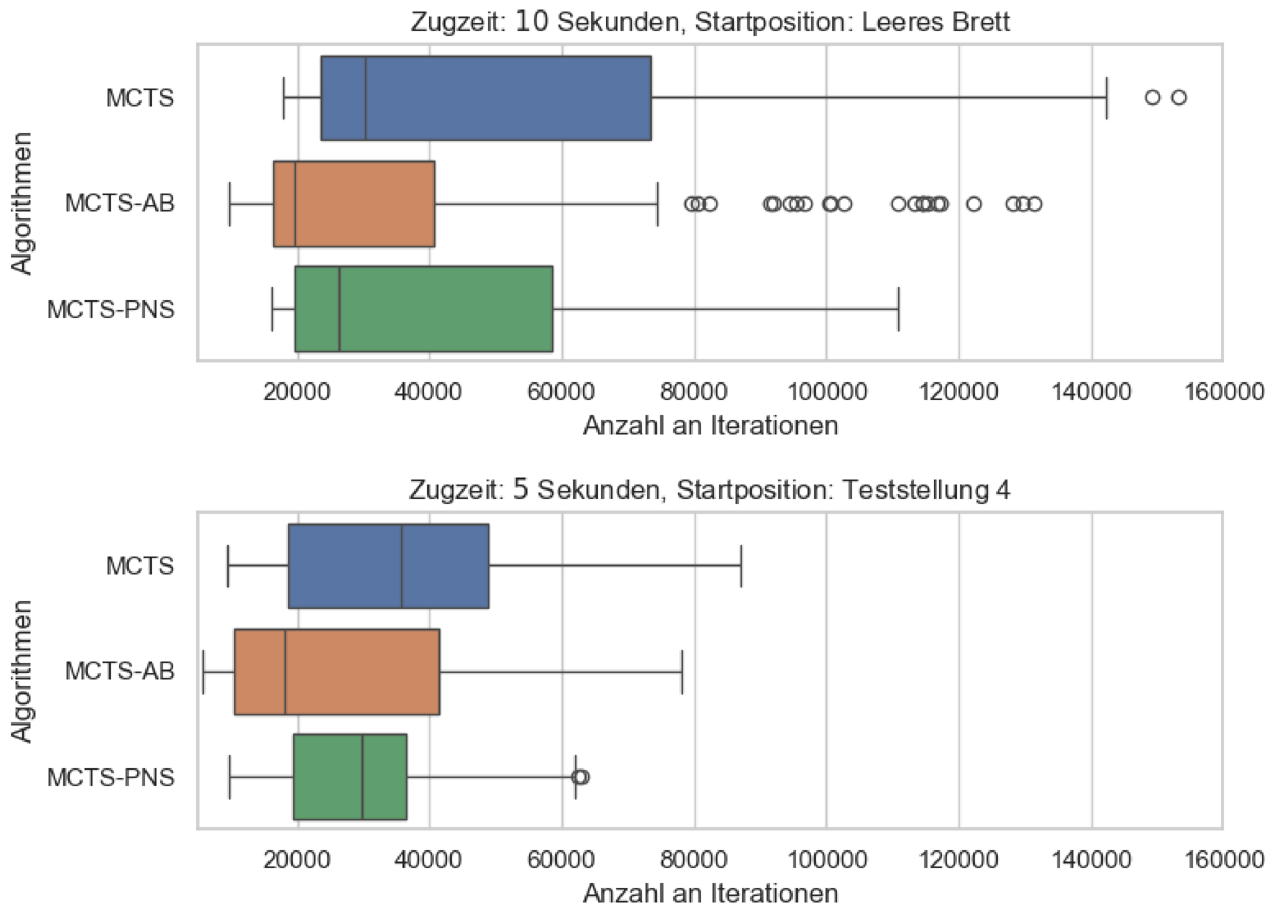


Abbildung 3.8: Durchschnittliche Iterationen der MCTS Varianten pro Zug

3.3.2 Untersuchte Knoten

Für die MCTS-Algorithmen wurde die durchschnittliche Anzahl an untersuchten Knoten erfasst. Auch hier entspricht jeder Datenpunkt dem Durchschnitt eines einzelnen Spiels. Dabei zählt jeder expandierte Knoten als untersucht.

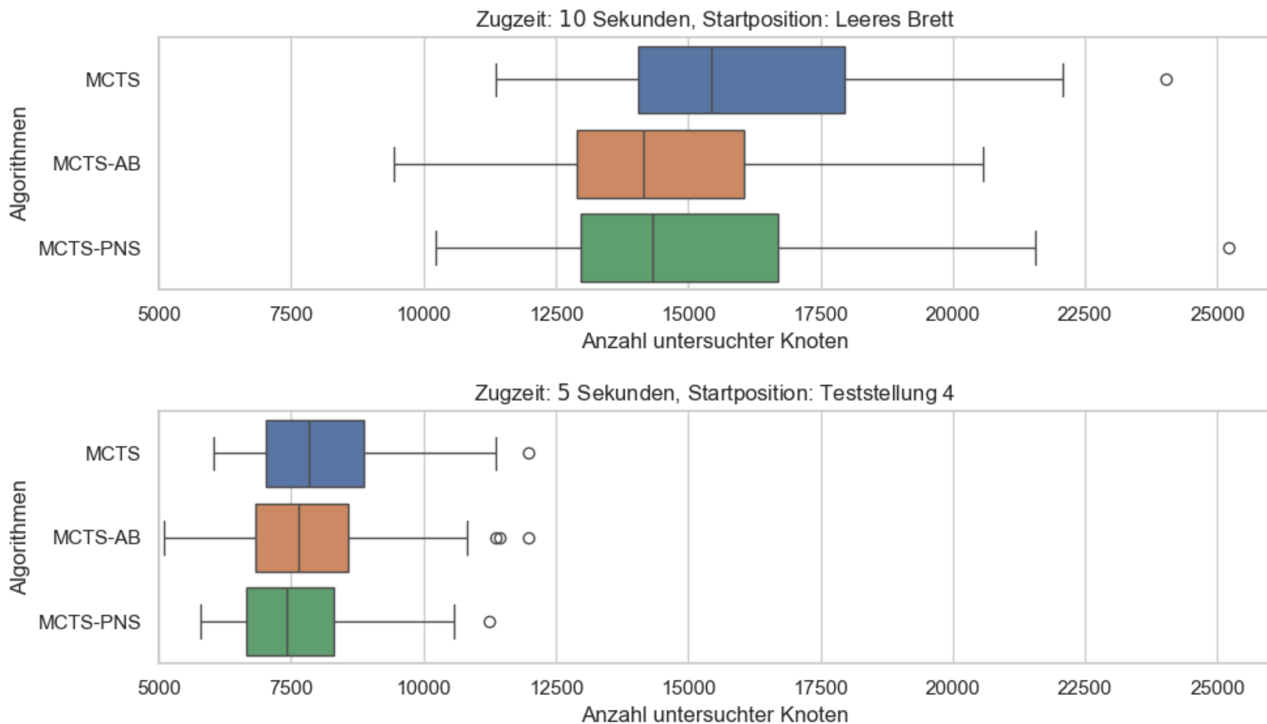


Abbildung 3.9: Durchschnittliche Anzahl an untersuchten Knoten pro Zug

3.3.3 Speicherbedarf der Transpositionstabelle

Während der durchgeführten Spiele hatte der Transpositionstabelle eine durchschnittliche Menge von 68.110 Einträgen[2]. Mit einer Größe von 88 Bytes [3] pro Eintrag wurde eine durchschnittliche Größe der Tabelle von $5,72$ MB bestimmt.
Für die Tests mit 5 Sekunden Zeit pro Zug sind es im Schnitt 35757 Einträge und somit eine Größe von 3 MB.

3.3.4 Ergebnisse der Spiele

Für die Berechnung der Siegesraten wurden Spiele die in einem Unentschieden endeten für beide als Niederlage gewertet. Die Siegesraten zwei gegenüberstehender Algorithmen müssen zusammenaddiert also nicht unbedingt 100% ergeben.

$$\text{Siegesrate} = \frac{\text{Anzahl Siege}}{\text{Anzahl Spiele}}$$

[2]Diese Werte stammen nur aus den Spielen zwischen **AB** und **MCTS-AB**, auf Grund einer fehlerhaften Implementierung des Testes.
[3]Da die Größe von Datentypen in Python variabel sind, kann diese Angabe auf anderen Systemen abweichen.

Zugzeit: 10 Sekunden, Startposition: Leeres Brett				
Kandidat	Gegner			
	AB*	**MCTS**	**MCTS-AB**	**MCTS-PNS**
AB*	–	76	82	82
MCTS	24	–	46	38
MCTS-AB	16	46	–	44
MCTS-PNS	18	54	42	–

Tabelle 3.4: Siegesraten (in %, gerundet). Teil 1

Zugzeit: 5 Sekunden, Startposition: Teststellung 4				
Kandidat	Gegner			
	AB*	**MCTS**	**MCTS-AB**	**MCTS-PNS**
AB*	–	58	62	62
MCTS	36	–	52	52
MCTS-AB	28	40	–	50
MCTS-PNS	32	46	46	–

Tabelle 3.5: Siegesraten (in %, gerundet). Teil 2

Zugzeit	**AB***	**MCTS**	**MCTS-AB**	**MCTS-PNS**
$10s$	1 (0.3%)	8 (2.7%)	12 (4%)	11 (3.7%)
$5s$	11 (3.7%)	8 (2.7%)	11 (3.7%)	6 (2%)

Tabelle 3.6: Anzahl und Anteil der Spiele, welche in einem Unentschieden endeten.

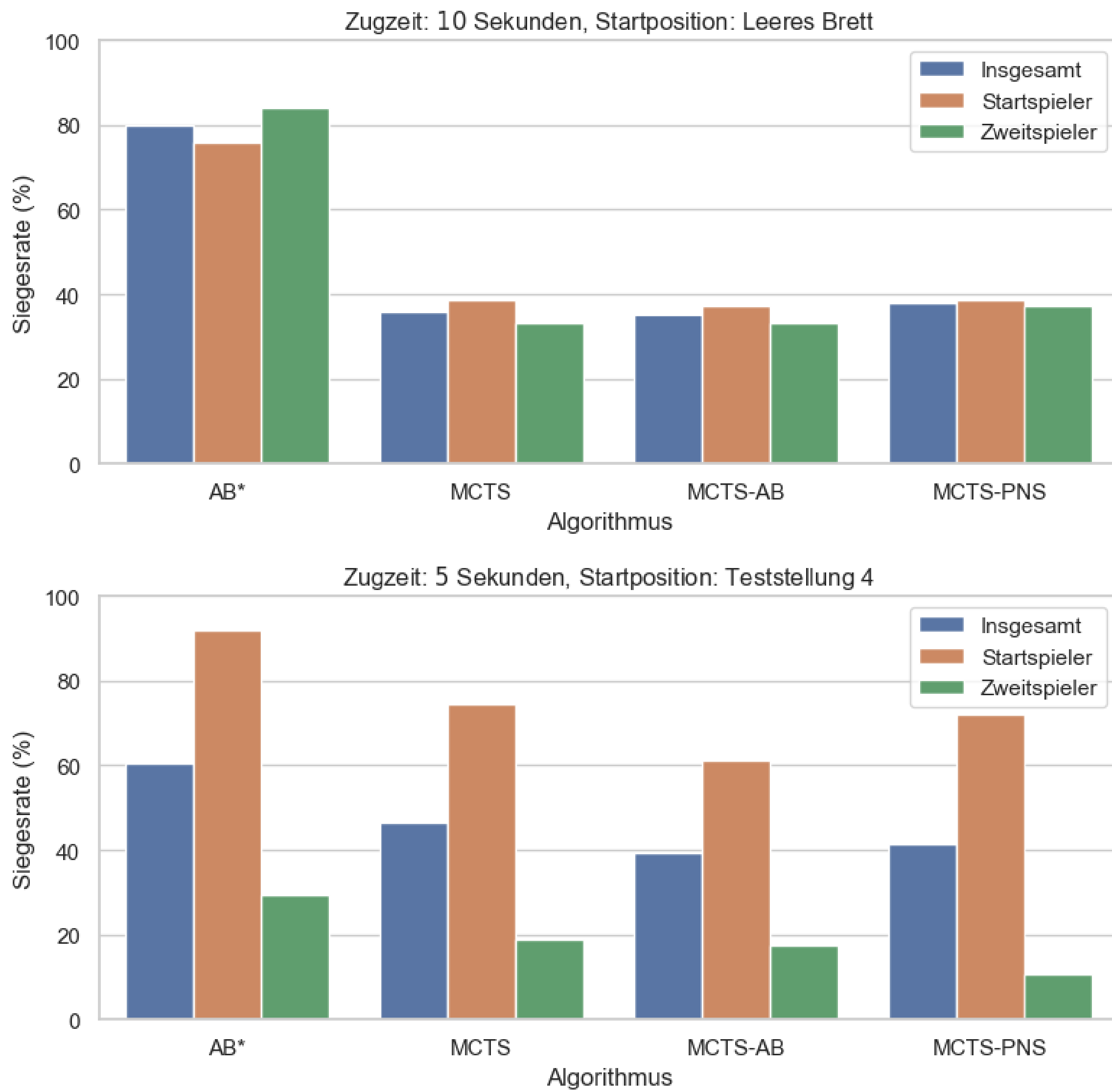
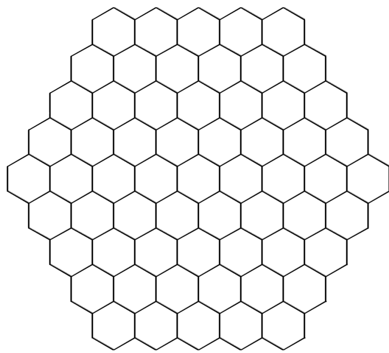
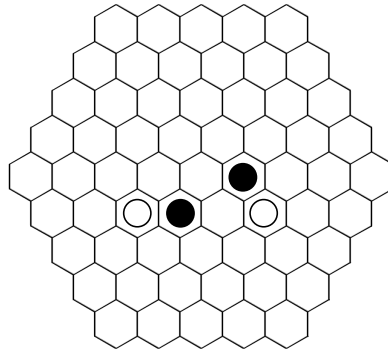


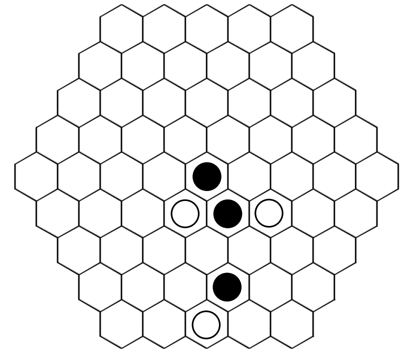
Abbildung 3.10: Siegesraten der Algorithmen im Round-Robin Verfahren



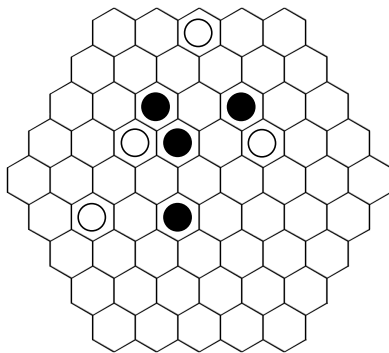
(a) Spielstellung 1



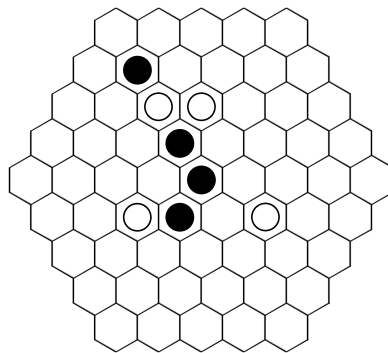
(b) Spielstellung 2



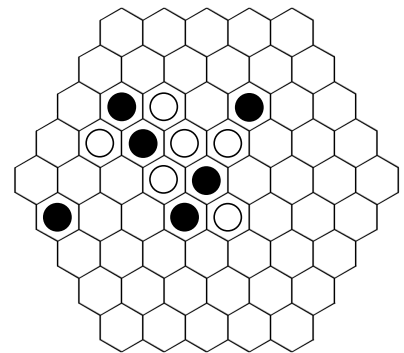
(c) Spielstellung 3



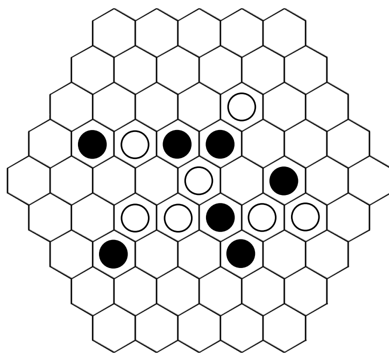
(d) Spielstellung 4



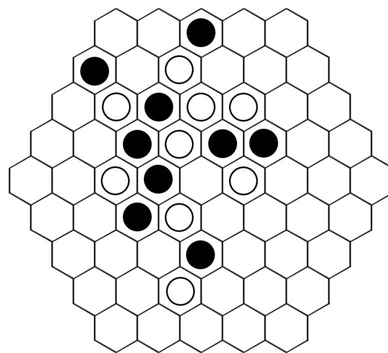
(e) Spielstellung 5



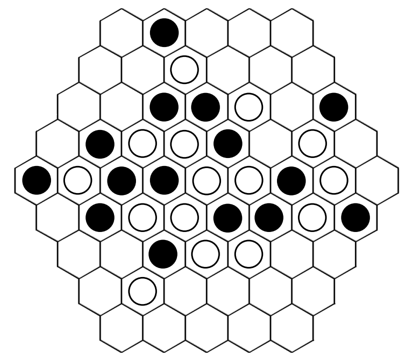
(f) Spielstellung 6



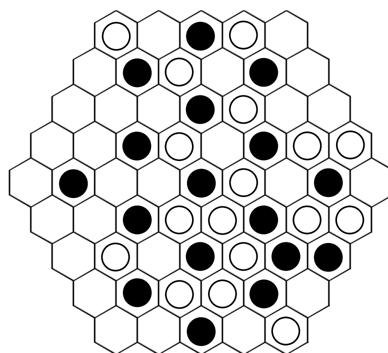
(g) Spielstellung 7



(h) Spielstellung 8



(i) Spielstellung 9



(j) Spielstellung 10

Abbildung 3.11: Spielstellungen für die Tests

4 Diskussion

4.1 Vergleich der Alpha-Beta Varianten

In Abbildung 3.1 ist für alle untersuchten Suchtiefen ein klarer Zusammenhang zwischen der Wahl des Algorithmus und der benötigten Berechnungszeit zu erkennen. Die Ergebnisse bestätigen, dass die Verwendung von Transpositionstabellen und iterativer Tiefensuche sich im Durchschnitt positiv auf die Laufzeit der Alpha-Beta Suche auswirkt, wobei Iterative Deepening eine größere Beschleunigung des Suchvorgangs verursacht. Diese Beobachtungen gehen einher mit den Erwartungen aus der Literatur. Korf [11] zeigt, dass die iterative Tiefensuche im Vergleich zu Brute-Force Ansätzen asymptotisch optimal hinsichtlich Laufzeit ist. Greenblatt et al. [8] beschrieben, dass die Verwendung einer Transpositionstabelle[1] zu einer Reduktion der Berechnungszeit führt, indem bereits untersuchte Spielstellung nicht erneut berechnet werden.

In den erhobenen Daten ist eine Ausnahme dieser Beobachtungen zu erkennen. Wie in Abbildung 3.2 zu sehen ist, sorgt die Nutzung von iterativer Tiefensuche in Teststellung 1, also dem leeren Brett, dafür dass sich die Berechnungszeit im Vergleich zu der jeweiligen Variante welche nur die Transpositionstabelle verwendet, bzw. ohne Modifikationen läuft, verlangsamt. Grund dafür ist, dass sich für das leere Brett der Mehraufwand niedrigere Suchtiefen zu berechnen nicht lohnt. Denn die Bewertungsfunktion kann dabei, durch die geringe Anzahl an gesetzten Steinen in frühen Suchtiefen, keine großen Unterschiede zwischen den Zügen machen und somit kann die Zugsortierung nicht verbessert werden. Ab der Suchtiefe 6 ist dieses Phänomen allerdings nicht mehr zu beobachten.

Auffällig ist außerdem dass die Zeiten für Teststellungen 2 und 3 deutlich höher sind als für Spielbrett 1. Da auf dem leeren Brett noch keine Fixpunkte existieren, gibt es viele Züge, welche zu Spielpositionen mit Mustern führen, die bereits in einer anderen Rotation, bzw. Position, untersucht wurden und somit alle gleich bewertet werden, sodass mehr alpha-beta cutoffs vorgenommen werden können. Für Stellungen 2 und 3 ist dies nicht der Fall, aber die Anzahl der gesetzten Steine reduziert den Suchraum noch nicht ausreichend, wie in den folgenden Testbrettern.

4.2 Vergleich der MCTS Varianten

4.2.1 Explorationskonstante c

Die Untersuchungen hinsichtlich der Explorationskonstante weisen darauf hin, dass im Kontext von Yavalath ein im Vergleich zu der in der Literatur üblichen Konstante $c = \sqrt{2}$ [10] [4][2] ein eher

[1] Im zitierten Paper hash table genannt.

[2] In der originalen Literatur wird von $c = \frac{1}{\sqrt{2}}$ gesprochen, aber wird in der dort beschriebenen UCT-Formel, im Gegensatz zu der in dieser Arbeit beschriebenen Formel, mit 2 multipliziert.

niedriger c-Wert von Vorteil ist. Dies ist ein Indikator dafür, dass in den meisten Spielsituationen nur wenige gute Züge zu finden sind, welche durch eine stärkere Exploitation deutlicher gemacht werden.

In Abbildung 3.6 kann man beobachten, dass kein linearer Zusammenhang zwischen der Höhe des c-Wertes und seiner Leistung steht. Dies folgt vermutlich daraus, dass im Durchschnitt für $c = 0.5$ in der gegebenen Zeit der Suchbaum nicht genug erforscht wird, um tatsächlich bessere Züge zu finden, aber für $c = 0.7$ das häufiger der Fall war.
Bei der Wiederholung dieses Tests hat sich die Siegesrate im direkten Vergleich von $c = 0.3$ und 0.7 (vgl. Tab. 3.2 und Tab. 3.3) nicht verändert, aber die Rate für 0.7 ist insgesamt gestiegen. Trotz des direkten Vergleiches wurde $c = 0.7$ für die folgenden Experimente gewählt, um die tatsächlich beste Explorationsrate bei einer höheren Zugzeit nicht zu unterschätzen.

4.2.2 General Domain Knowledge

Die Ergebnisse in Abbildung 3.5 zeigen klar, dass Monte Carlo Tree Search im Kontext von Yavlath stark von der Verwendung von General Domain Knowledge profitiert. Dies deckt sich mit den von Browne [2] beschriebenen Beobachtungen.

4.3 Vergleich der ausgewählten Algorithmen

Untersuchte Knoten

Wie zu erwarten werden durch **MCTS** deutlich mehr Knoten untersucht, als durch **MCTS-AB** und **MCTS-PNS**, da diese zusätzliche Berechnungen von Alpha-Beta-Suchen, bzw. der proof numbers, durchführen. Es kann ebenfalls ein klarer Zusammenhang zwischen den durchschnittlich untersuchten Knoten und der Berechnungszeit pro Zug beobachtet werden. Abbildung 3.9 zeigt die doppelte Anzahl an Knoten für die doppelte Berechnungszeit. Die Ergebnisse deuten also auf einen linearen Zusammenhang hin.

Iterationen

Während auch für die Anzahl der Iterationen der MCTS Varianten eine Reduktion der Extrema zwischen den Testdurchläufen erkennbar ist, bleiben im Allgemeinen die aufgezeichneten Mittelwerte in der gleichen Größenordnung, für **MCTS** und **MCTS-PNS** ist sogar eine Zunahme des Medians zu beobachten. Dieses Phänomen muss durch die Änderung der Startposition verursacht worden sein, da dort die frühen Spielzüge mit den meisten Expansionen des Suchbaumes nicht Teil der Datenmenge sind.

Ergebnisse der Spiele

In den durchgeführt Experimenten konnte klar gezeigt werden, dass **AB*** den anderen Algorithmen bezüglich der Spielstärke überlegen war. Der Nachteil, dass Monte Carlo Tree Search lediglich in polynomieller Zeit gegen die optimale Lösung konvergiert [4], konnte somit im Rahmen dieser Experimente deutlich beobachtet werden.

Im direkten Vergleich unter den MCTS Varianten konnte beobachtet werden, dass **MCTS-PNS** mit ausreichender Zeit die Standardimplementierung schlägt. Auch für **MCTS-AB** konnte eine Erhöhung der Siegesrate mit zunehmender Zeit in diesem direkten Vergleich festgestellt werden. Es existiert also eine gewisse Schwelle in der Berechnungszeit, ab der sich der Rechenaufwand zur Behandlung von sudden death und sudden win Situationen lohnt.

4.4 Limitationen

Die Aussagekraft der Erkenntnisse dieser Arbeit können durch eine weiterführende Analyse der Algorithmen bezüglich größeren Rechenkapazitäten verstärkt werden. Mögliche Änderungen der Rechenkapazität beinhalten die effizientere Implementierung der Methoden, z.B. in einer anderen Programmiersprache als Python, und weitere Tests mit erhöhter Berechnungszeit pro Zug. Im Rahmen dieser Arbeit wurden also vielmehr Tendenzen gefunden, als vollständige Zusammenhänge erkannt. Das weitere Erproben der Algorithmen in einzelnen Spielsituationen könnten weitere Stärken und Schwächen aufzeigen. Außerdem sind weitere Optimierungen der Algorithmen möglich. Zum Beispiel würde eine Überarbeitung der Bewertungsfunktion für eine stärkere Alpha-Beta-Suche in Betracht gezogen werden, da so auch die hybride MCTS-AB Methode verbessert werden würde.

4.5 Beitrag der Arbeit

Durch den systematischen Vergleich von den untersuchten Algorithmen konnten die Erwartungen aus der Literatur im Kontext Yavalath bestätigt werden. Es wurden zwei Modifikationen des Monte Carlo Tree Search vorgestellt und experimentell erprobt. Auf dieser Grundlage können in Zukunft zusätzliche Analysen dieser Methoden aufgebaut werden, welche in dieser Arbeit als vielversprechend wahrgenommen wurden.

5 Fazit

Im Rahmen dieser Arbeit wurden die Alpha-Beta-Suche, Monte Carlo Tree Search und zwei MCTS-Hybride mit Alpha-Beta und Proof Number Search hinsichtlich ihrer Anwendung in dem Spiel Yavalath untersucht. Dabei war es das Ziel zu betrachten, wie die geringe domänenspezifischen Wissenslage über Yavalath die Leistung der Alpha-Beta-Suche beeinflusst und die Problematik von plötzlichen Spielenden das Verhalten von MCTS beeinträchtigt.

Trotz einer simplen Bewertungsfunktion stellt sich die Alpha-Beta-Suche als eine vergleichsweise gute Methode zur algorithmischen Zugfindung heraus. Des Weiteren konnten die Grenzen des Monte Carlo Tree Search experimentell beobachtet werden, wobei die Problematik von plötzlichen Spielenden die benötigte Zeit, um optimale Züge zu finden, erhöhte. Die untersuchten Modifikationen von MCTS mit der Alpha-Beta Suche und der Proof Number Search stellen zwei vielversprechende Optimierungen hinsichtlich dieser Herausforderung dar.

In der Zukunft können eine effizientere Implementierung oder weitere Optimierungen dazu beitragen die Leistung des MCTS weiter an die der Alpha-Beta-Suche anzunähern. Außerdem könnten andere Modifikationen von MCTS, wie zum Beispiel die gleichzeitige Verwendung von Alpha-Beta und Proof Number Search, untersucht werden.

Hilfsmittel

Zur Überprüfung und Überarbeitung der Rechtschreibung, Grammatik und Satzstruktur habe ich die generische KI ChatGPT (Model GPT-4.1-mini) von OpenAI verwendet.

Abbildungen, die nicht durch Code generiert wurden, wurden von mir mit Word und GIMP erstellt.

Literaturverzeichnis

[1] Dennis Breuker, Jos Uiterwijk, and H. Jaap van den Herik. Information in transposition tables. 8:1–13, 02 1970.

[2] Cameron Browne. *Automatic Generation and Evaluation of Recombination Games*. PhD thesis, Queensland University of Technology, 2008.

[3] Cameron Browne. Bitboard methods for games. *ICGA journal*, 37(2):67–84, 2014.

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

[5] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217, 2008.

[6] Elliot Doe, Mark HM Winands, Dennis JNJ Soemers, and Cameron Browne. Combining monte-carlo tree search with proof-number search. In *2022 IEEE Conference on Games (CoG)*, pages 206–212. IEEE, 2022.

[7] Daniel James Edwards and TP Hart. The alpha-beta heuristic. 1961.

[8] Richard D Greenblatt, Donald E Eastlake III, and Stephen D Crocker. The greenblatt chess program. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 801–810, 1967.

[9] Dominikus Herzberg. Bitboards and Connect Four. https://github.com/denkspuren/BitboardC4/blob/master/BitboardDesign.md. Accessed: 07.05.2025.

[10] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[11] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[12] William Li et al. Prediction distortion in monte carlo tree search and an improved algorithm. *Journal of Intelligent Learning Systems and Applications*, 10(2):46–79, 2018.

[13] Chess Programming Wiki. Alpha-Beta. https://www.chessprogramming.org/Alpha-Beta. Accessed: 18.05.2025.

[14] Chess Programming Wiki. Node Types. https://www.chessprogramming.org/Node_Types#PV. Accessed: 19.05.2025.

[15] Chess Programming Wiki. Transposition Table. https://www.chessprogramming.org/Transposition_Table. Accessed: 19.05.2025.

[16] Albert L. Zobrist. A new hashing method with application for game playing. Technical report, The University of Wisconsin, 1970.