Ein systematischer Vergleich von Alpha-Beta-Suche vs. MCTS im Spiel Quoridor

Bachelorarbeit

Tra My Kohlmann
484796

31. März 2026

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

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

Kurzfassung

Zwei-Personen-Nullsummenspiele gehören zu den klassischen Anwendungsfeldern von Suchalgorithmen in der Künstlichen Intelligenz. Die vorliegende Arbeit untersucht systematisch verschiedene Varianten der Alpha-Beta-Suche und von Monte Carlo Tree Search (MCTS) im Spiel Quoridor. Auf dieser Grundlage werden anschließend die jeweils leistungsstärksten Varianten bei vergleichbarem Rechenbudget direkt miteinander verglichen. Obwohl Quoridor auf relativ einfachen Regeln basiert, ist das Spiel algorithmisch anspruchsvoll. Dies liegt vor allem am hohen Verzweigungsfaktor und an der großen Anzahl möglicher Spielverläufe.

Für die Alpha-Beta-Suche wurden verschiedene Konfigurationen mit iterativer Tiefensuche, Transpositionstabelle und Zugsortierung untersucht. Für MCTS standen Basis-MCTS, heuristikbasierte Bewertung und eine strukturierte Playout-Policy im Mittelpunkt. Die jeweils leistungsstärksten Varianten wurden anschließend unter identischen Bedingungen direkt miteinander verglichen.

Die Ergebnisse zeigen, dass AB+ID+MO+TT unter den gewählten Versuchsbedingungen insgesamt die stärkeren Resultate erzielt. Insbesondere bei kurzen Zeitlimits weist Alpha-Beta Vorteile auf. Die heuristikbasierte MCTS-Variante profitiert hingegen stärker von zusätzlicher Rechenzeit und kann bei größeren Zeitbudgets aufschließen beziehungsweise in einzelnen Fällen bessere Ergebnisse erzielen. Insgesamt verdeutlichen die Ergebnisse, dass die relative Stärke beider Verfahren nicht nur vom Suchansatz selbst, sondern auch vom verfügbaren Rechenbudget sowie von der konkreten Implementierung abhängt.

Inhaltsverzeichnis

1 Einleitung

Deterministische Zwei-Personen-Nullsummenspiele bilden seit langem ein wichtiges Anwendungsfeld der Künstlichen Intelligenz. Besonders relevant sind dabei Verfahren der Spielbaumsuche. Sie ermöglichen es, mögliche Züge und ihre Folgen systematisch zu analysieren oder simulationsbasiert abzuschätzen. Zu den bekanntesten Ansätzen zählen die Alpha-Beta-Suche als klassisches tiefensuchbasiertes Verfahren und die Monte Carlo Tree Search als simulationsbasierte Methode.

Alpha-Beta-Suche durchsucht den Spielbaum systematisch bis zu einem begrenzten Suchhorizont. Nicht vollständig durchsuchte Positionen werden dabei mithilfe heuristischer Funktionen bewertet. Die Leistungsfähigkeit des Verfahrens hängt daher vor allem von der erreichten Suchtiefe und der Qualität der Heuristik ab. MCTS kombiniert dagegen selektive Baumexploration mit stochastischen Simulationen. Der Suchprozess wird dabei adaptiv über das Verhältnis von Exploration und Exploitation gesteuert. Als Spieldomäne dient das strategische Brettspiel Quoridor. Trotz seiner einfachen Regeln weist es durch die Kombination aus Figurenbewegung und Wandplatzierung einen vergleichsweise hohen Verzweigungsfaktor auf.

Ziel dieser Arbeit ist der systematische Vergleich verschiedener Varianten der Alpha-Beta-Suche und der Monte Carlo Tree Search im Spiel Quoridor. Zunächst werden die Varianten beider Verfahren untereinander evaluiert. Auf dieser Grundlage wird für jedes Verfahren die leistungsstärkste Konfiguration bestimmt. Anschließend werden die besten Ansätze unter identischen Bedingungen direkt miteinander verglichen. Vor diesem Hintergrund stellt sich die Frage, welche der beiden Suchmethoden in Quoridor bei vergleichbarem Rechenbudget und in ihrer jeweils leistungsstärksten Variante die höhere Spielstärke erzielt.

1.1 Quoridor-Regeln

Quoridor ist ein abstraktes Strategiespiel für zwei oder vier Spieler, das vom Schweizer Spieleautor Mirko Marchesi entwickelt und vom Verlag Gigamic veröffentlicht wurde [1]. Im Rahmen dieser Arbeit wird ausschließlich die Zwei-Personen-Variante betrachtet. Ziel des Spiels ist es, die eigene Spielfigur als Erste auf die gegenüberliegende Grundlinie des 9×9-Spielfelds zu bewegen. Dabei können Wände strategisch gesetzt werden, um den Weg des Gegners zu blockieren oder zu verlängern.

Die Spielidee geht auf das ältere Brettspiel Blockade (auch Cul-de-sac) zurück, das 1975 von Philip Slater entworfen wurde. Dort bewegen sich Spielfiguren bereits über ein Gitter mit Barrieren. Darauf aufbauend entwickelte Marchesi zunächst Pinko Pallino (1995) und später Quoridor (1997). Mit seinem 9×9-Spielfeld und den frei platzierbaren Wänden entspricht Quoridor der heute verbreiteten

[1] https://de.wikipedia.org/wiki/Quoridor, Zugriff: 20.02.2026.

Form des Spiels.[2].

Jeder Spieler startet mit einem Spielstein, der zu Beginn auf dem mittleren Feld der eigenen Grundlinie platziert wird (vgl. Abb. 1.1). Das Ziel besteht darin, diesen Stein als erstes auf die gegenüberliegende Grundlinie zu bewegen.

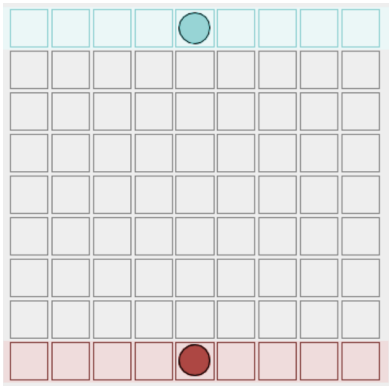


Abbildung 1.1: Startaufstellung [9]

Zu Beginn stehen jedem Spieler 10 Zäune zur Verfügung. Diese werden zwischen zwei Felderreihen platziert und grenzen jeweils an vier Felder, davon zwei auf jeder Seite. Einmal gesetzte Zäune können weder bewegt noch entfernt werden. Pro Zug entscheidet ein Spieler, ob er seine Figur bewegt oder einen Zaun setzt. Sind alle Zäune verbraucht, muss die Figur bewegt werden.

Zäune können eingesetzt werden, um den eigenen Weg zur Zielreihe abzusichern oder den Weg des Gegners zu verlängern. Es ist jedoch unzulässig, den Pfad eines Spielers zur gegnerischen Grundlinie vollständig zu versperren. Es muss stets mindestens ein freier Weg verbleiben.

Spielfiguren bewegen sich pro Zug um genau ein Feld waagerecht oder senkrecht. Stehen sich zwei Figuren auf benachbarten Feldern ohne trennenden Zaun gegenüber, darf die ziehende Figur über die gegnerische Figur springen. Sie rückt dann auf das direkt dahinterliegende Feld vor. Befindet sich hinter der gegnerischen Figur ein Zaun, ist dieser Sprung nicht möglich. Stattdessen darf die eigene Figur auf ein seitlich benachbartes Feld ausweichen (vgl. Abb. 1.2) [3].

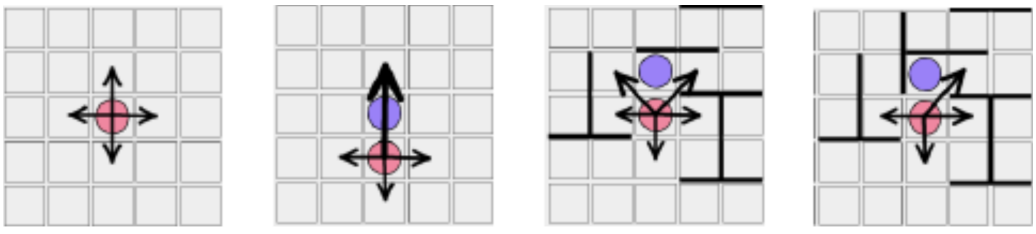


Abbildung 1.2: Zulässige Züge für den unteren roten Stein [10]

[2] https://en.wikipedia.org/wiki/Quoridor, Zugriff: 20.02.2026.
[3] https://www.brettspielversand.de/mediafiles/spieleanleitungen/gigamic/155-0019_Quoridor_Anleitung.pdf, Zugriff: 20.02.2026.

1.2 Komplexität

Für die Analyse der algorithmischen Anforderungen eines Quoridor-Agenten ist zunächst ein Blick auf die spieltheoretische Komplexität des Spiels sinnvoll. Dabei sind vor allem zwei Kenngrößen relevant. Die Zustandsraumkomplexität beschreibt die Anzahl aller möglichen Spielstellungen, während die Spielbaumkomplexität die Anzahl aller potenziell spielbaren Partien angibt.

Die Zustandsraumkomplexität von Quoridor ergibt sich aus der Kombination möglicher Figurenpositionen und zulässiger Wandplatzierungen. In der Zwei-Personen-Variante stehen für die erste Spielfigur 81 und für die zweite 80 Felder zur Verfügung. Daraus ergeben sich insgesamt 6.480 mögliche Figurenkonfigurationen. Hinzu kommen horizontale und vertikale Wandplatzierungen. Bestimmte Kombinationen sind jedoch durch die Spielregeln ausgeschlossen. Unter Berücksichtigung aller zulässigen Konfigurationen schätzt Mertens (2006) [7] die Zustandsraumgröße auf eine Größenordnung von 10^{42}.

Die Spielbaumkomplexität wird anhand des durchschnittlichen Verzweigungsfaktors und der mittleren Spiellänge abgeschätzt. Für Quoridor wird ein Verzweigungsfaktor von etwa 60 möglichen Zügen pro Stellung angenommen. Die mittlere Spieldauer liegt bei rund 90 Halbzügen. Daraus ergibt sich eine Spielbaumkomplexität in der Größenordnung von 10^{162}.

Ein Vergleich mit etablierten Brettspielen wie Schach oder Go (vgl. Abb. 1.3) zeigt, dass die Zustandsraumkomplexität von Quoridor in einer ähnlichen Größenordnung wie die von Schach liegt. Die Spielbaumkomplexität ist dagegen zum Teil sogar höher. Daher wird Quoridor häufig der höchsten Komplexitätsklasse schwer lösbarer Spiele zugeordnet [7, 9].

Game	log(state-space)	log(game-tree)
Tic-tac-toe	3	5
Nine Men's Morris	10	50
Awari/Oware	16	32
Pentominoes	12	18
Connect Four	14	21
Checkers	33	50
Lines of Action	24	56
Othello	28	58
Backgammon	20	144
Quoridor	42	162
Chess	46	123
Xiangqi	52	150
Arimaa	42	190
Shogi	71	226
Connect6	172	140
Go	172	360

Abbildung 1.3: Spielkomplexität [7]

1.3 Literatur

Im Vergleich zu Schach oder Go, die algorithmisch umfassend erforscht sind, ist Quoridor bislang deutlich weniger untersucht. Fundierte Erkenntnisse über optimale Gewinnstrategien sind kaum ver-

fügbar. Die wenigen vorhandenen Arbeiten konzentrieren sich dabei überwiegend auf klassische Spielbaumsuchverfahren.

Glendenning (2005) [3] implementierte in Mastering Quoridor ein iteratives Deepening-Verfahren auf Basis von Alpha-Beta-Negamax. Die Bewertungsfunktion kombinierte dabei heuristische Merkmale wie die kürzeste Zieldistanz, die Manhattan-Distanz, Markov-Chain-basierte Abschätzungen sowie die Anzahl verbleibender Wände. Die Gewichtungsparameter wurden zusätzlich mithilfe eines genetischen Algorithmus optimiert. Der Agent erwies sich jedoch als vergleichsweise schwach und ließ sich von menschlichen Spielern relativ leicht besiegen.

Mertens (2006) [7] entwickelte einen Agenten auf Basis von Minimax mit Alpha-Beta-Pruning, bei dem ebenfalls eine tiefenbegrenzte Suche mit einer heuristischen Bewertungsfunktion kombiniert wurde. Auch dieser Agent blieb leistungsschwach und konnte langfristige strategische Zusammenhänge nur begrenzt berücksichtigen.

Respall et al. (2018) [10] stellten einen MCTS-Agenten für Quoridor vor und verglichen diesen mit mehreren anderen Agententypen. MCTS erwies sich als probabilistischer Suchalgorithmus grundsätzlich als geeignet, mit dem hohen Verzweigungsfaktor des Spiels umzugehen. In der Simulationsphase wurde eine heuristische Komponente integriert. Dadurch konnte das Verhältnis zwischen Wandplatzierung und Figurenbewegung gezielter gesteuert werden. Die experimentellen Ergebnisse erlauben keine abschließende Bewertung der Spielstärke. Sie geben jedoch eine erste Einschätzung der Leistungsfähigkeit gegenüber menschlichen Spielern. Auffällig ist, dass der Agent mit 60.000 Simulationen bessere Ergebnisse erzielte als jener mit 120.000 Simulationen. Das spricht dafür, dass die Qualität der Simulationen und die Ausgestaltung der Heuristik wichtiger für die Spielstärke sind als die reine Anzahl der Durchläufe.

1.4 Struktur der Arbeit

Kapitel 1 führt in das Thema ein, beschreibt die Spielregeln und gibt einen Überblick über die historische Entwicklung von Quoridor. Anschließend werden die spieltheoretische Komplexität und verwandte Arbeiten vorgestellt.

Kapitel 2 behandelt die methodischen Grundlagen der untersuchten Algorithmen. Zunächst werden Alpha-Beta-Suche, die Bewertungsfunktion und Strategien zur Effizienzsteigerung vorgestellt. Danach wird MCTS vom Basisalgorithmus über UCT bis hin zu weiteren Erweiterungen behandelt.

Kapitel 3 präsentiert die experimentellen Ergebnisse. Nach der Beschreibung der Testbedingungen und der eingesetzten Hardware werden die Alpha-Beta-Such- und MCTS-Varianten hinsichtlich Berechnungszeit, Suchtiefe und Spielstärke evaluiert. Abschließend werden die stärksten Varianten beider Verfahren in einem direkten Vergleich gegenübergestellt.

Kapitel 4 interpretiert die experimentellen Ergebnisse beider Suchalgorithmen, diskutiert auffällige Befunde und erörtert die Limitationen der Arbeit.

Kapitel 5 fasst die zentralen Erkenntnisse zusammen, beantwortet die Forschungsfrage und stellt mögliche Ansätze für weiterführende Arbeiten vor.

2 Methoden

2.1 Alpha-Beta-Suche

Die im vorherigen Abschnitt dargestellte Spielbaumkomplexität von Quoridor macht deutlich, dass die Wahl eines geeigneten Suchalgorithmus bedeutend ist. Ein in strategischen Zwei-Personen-Spielen weit verbreiteter Ansatz ist die Minimax-Suche mit Alpha-Beta-Pruning.

Alpha-Beta-Suche (engl. Alpha-Beta-Pruning) ist eine Optimierung der Minimax-Suche. Sie reduziert die Anzahl der zu durchsuchenden Knoten. Das Ergebnis wird dadurch nicht verändert. Teilbäume werden verworfen, sobald ihre weitere Untersuchung keinen Einfluss mehr auf die endgültige Entscheidung hat. Dabei verwaltet jeder Knoten zwei Schrankenwerte. Der Wert α repräsentiert die beste bekannte untere Schranke für den MAX-Spieler, während β die beste bekannte obere Schranke für den MIN-Spieler angibt. In MAX-Knoten wird α aktualisiert, sobald ein Nachfolger einen höheren Bewertungswert liefert. In MIN-Knoten wird β entsprechend angepasst. Gilt für einen Knoten die Bedingung $\alpha \geq \beta$, kann der zugehörige Teilbaum verworfen werden.

Typischerweise wird Alpha-Beta-Suche als Tiefensuche bis zu einer vorgegebenen Suchtiefe umgesetzt. Dabei wird an MAX-Knoten das Maximum und an MIN-Knoten das Minimum der Nachfolgebewertungen bestimmt. Durch das frühzeitige Verwerfen irrelevanter Teilbäume kann die Anzahl zu evaluierender Knoten reduziert werden [11].

2.1.1 Bewertungsfunktion

In der Alpha-Beta-Suche bezeichnet der Begriff Heuristik eine Bewertungsfunktion. Sie schätzt eine Spielsituation näherungsweise ein, ohne die Partie bis zu einem terminalen Zustand vollständig auswerten zu müssen. In komplexen Spielen wächst der Suchbaum exponentiell an. Daher wird die Suche nach einer vorgegebenen Tiefe abgebrochen und die erreichte Stellung heuristisch bewertet.

Diese Bewertung orientiert sich an spielspezifischen Merkmalen und liefert dem Algorithmus eine Schätzung der späteren Gewinnchancen. Da Quoridor ein Zwei-Personen-Nullsummenspiel ist, muss die Bewertungsfunktion diese Eigenschaft widerspiegeln, sodass ein Vorteil für einen Spieler als entsprechenden Nachteil für den Gegenspieler bewertet wird. Der Stellungswert wird meist als gewichtete Summe mehrerer Merkmale berechnet, die unterschiedliche Aspekte der aktuellen Spielsituation erfassen (vgl. Formel 2.1) [11].

$$\mathrm{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{2.1}$$

Dabei bezeichnet $f_i(s)$ das i-te Bewertungsmerkmal der Spielstellung s und w_i dessen Gewicht.

Bislang liegt nur wenig Forschung zu Quoridor vor. Daher existieren keine standardisierten Bewertungsmerkmale für die heuristische Stellungsbewertung. Zudem ist weitgehend unklar, welche Merkmale sich in der Praxis als geeignet erweisen. In der vorhandenen Literatur schlagen einzelne Autoren verschiedene Bewertungsmerkmale vor, darunter Glendenning (2005) [3], Mertens (2006) [7] sowie Dais und Prentzas[1]. Die vorgeschlagenen Merkmale überschneiden sich dabei teilweise.

Eines der grundlegendsten Bewertungsmerkmale ist die Distanz des Spielsteins zur Zielreihe (*distance_win*). Sie entspricht der Länge des kürzesten Pfades vom aktuellen Feld bis zur Zielreihe. Da platzierte Wände direkte Bewegungen blockieren können, wird diese Distanz mittels Breitensuche (*Breadth-First Search*) bestimmt.

Eng damit verknüpft ist das Merkmal *row_progress*. Es beschreibt die Differenz der zielbezogenen Zeilenfortschritte beider Spieler.

Spieler versuchen häufig, durch gezielte Wandplatzierungen den direkten Vorwärtsweg des Gegners zu blockieren und Ausweichbewegungen zu erzwingen. Daraus ergibt sich das Merkmal *distance_next*. Es berechnet die minimale Anzahl an Zügen, die erforderlich ist, um die nächste Reihe in Richtung der Zielreihe zu erreichen (vgl. Abb. 2.1).

Das vierte Merkmal erfasst die Anzahl der noch verfügbaren Wände pro Spieler (*walls*).

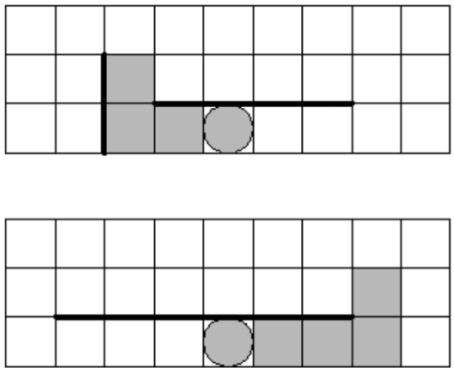


Abbildung 2.1: Minimale Anzahl an Zügen bis zur nächsten Reihe in Zielrichtung [7]

Zusammenfassend ergibt sich aus den beschriebenen Merkmalen $f_1, \ldots, f_4$ die Bewertungsfunktion gemäß Gleichung (2.1).

$$\mathrm{Eval}(s) = \sum_{i=1}^{n} w_i f_i(s)$$

- f_1 (*distance_win*): Distanz des Spielsteins zur Zielreihe
- f_2 (*row_progress*): Differenz des Spielfortschritts beider Figuren in Richtung der Zielreihe
- f_3 (*distance_next*): minimale Zuganzahl bis zur nächsten Reihe in Zielrichtung
- f_4 (*walls*): Anzahl der verfügbaren Wände pro Spieler

[1] https://github.com/pavlosdais/Quoridor, Zugriff: 21.02.2026

Die verwendeten Gewichte basieren auf den von Dais und Prentzas[2] vorgeschlagenen Werten und sind in Tabelle 2.1 aufgeführt.

Merkmal	Gewicht	Wert
f_1	w_1	1,0
f_2	w_2	0,1
f_3	w_3	0,6
f_4	w_4	0,7

Tabelle 2.1: Gewichte der Bewertungsmerkmale

Zusätzlich zu den beschriebenen Bewertungsmerkmalen wird ein sogenannter Tie-Breaker eingesetzt, der in Endspielsituationen relevant ist. Da viele Züge durch die Bewertungsfunktion ähnlich eingeschätzt werden, kann es zu unklaren Entscheidungen kommen.

Der Tie-Breaker erfüllt zwei Funktionen. Bei einer verbleibenden Distanz von höchstens fünf Zügen zur Zielreihe wird die Differenz der Zieldistanzen beider Spieler höher gewichtet. Dadurch beeinflussen bereits kleine Unterschiede in der Pfadlänge die Entscheidung. Bei einer Restdistanz von höchstens drei Zügen wird zusätzlich ein stark gewichteter Bonus bzw. eine Strafkomponente eingeführt. So wird der Algorithmus in Gewinnsituationen konsequent auf den Spielabschluss ausgerichtet und drohende Niederlagen werden berücksichtigt.

Algorithm 1 Bewertungsfunktion mit Endspiel-Tie-Breaker

score $\leftarrow (black\_distance\_win - white\_distance\_win)$
score $\leftarrow score + 0.6 \cdot (black\_distance\_next - white\_distance\_next)$
score $\leftarrow score + 0.7 \cdot (white\_walls - black\_walls)$
score $\leftarrow score + 0.1 \cdot (white\_row\_progress - black\_row\_progress)$

if $white\_distance\_win \leq 5$ **or** $black\_distance\_win \leq 5$ **then**
    score $\leftarrow score + 2 \cdot (black\_distance\_win - white\_distance\_win)$
    if $white\_distance\_win \leq 3$ **then**
        score $\leftarrow score + 20 \cdot (4 - white\_distance\_win)$
    end if
    if $black\_distance\_win \leq 3$ **then**
        score $\leftarrow score - 20 \cdot (4 - black\_distance\_win)$
    end if
end if
if not player_one_maximizer **then**
    score $\leftarrow -score$
end if
return score

[2] https://github.com/pavlosdais/Quoridor, Zugriff: 21.02.2026

2.1.2 Strategien zur Effizienzsteigerung der Alpha-Beta-Suche

Zwar reduziert die Alpha-Beta-Suche die Anzahl der zu betrachtenden Knoten gegenüber der klassischen Minimax-Suche. Die tatsächliche Effizienz hängt jedoch von zusätzlichen Verfahren ab. Diese können den Suchaufwand reduzieren, die Zugreihenfolge verbessern oder die Suche unter einer Zeitbegrenzung praktikabler machen. Die in dieser Arbeit eingesetzten Verfahren werden im Folgenden vorgestellt.

2.1.2.1 Iterative Tiefensuche

Die iterative Tiefensuche (*engl. iterative deepening*) kombiniert Elemente der Tiefen- und Breitensuche. Dabei wird die Alpha-Beta-Suche wiederholt mit schrittweise wachsender Suchtiefe ausgeführt. Die Suche beginnt bei Tiefe eins. Danach wird die maximale Suchtiefe schrittweise erhöht, bis entweder eine vorgegebene Maximaltiefe erreicht oder ein Zeitlimit überschritten wird. [5].

Für den Vergleich mit MCTS wird eine feste Zeitbegrenzung verwendet, da auch dieses Verfahren unter einer Zeitbeschränkung arbeitet. Zum besseren Verständnis ist ein Vergleich mit klassischen Suchstrategien hilfreich. Die Tiefensuche verfolgt jeden Pfad bis zur maximalen Tiefe. Bei einem Suchbaum mit Verzweigungsfaktor b und Tiefe d ergibt sich eine Zeitkomplexität von $O(b^d)$. Der Speicherbedarf beträgt lediglich $O(d)$, weil nur der aktuelle Pfad gespeichert wird. Die Breitensuche expandiert hingegen alle Knoten einer Ebene vollständig, bevor sie in die nächste übergeht. Zwar liegt die Zeitkomplexität ebenfalls bei $O(b^d)$, der Speicherbedarf ist jedoch mit $O(b^d)$ höher.

Die iterative Tiefensuche kombiniert die Vorteile beider Verfahren, indem sie wiederholt Tiefensuchen mit schrittweise wachsender Tiefe durchführt. Obwohl obere Ebenen dabei mehrfach untersucht werden, bleibt die Gesamtzeitkomplexität bei $O(b^d)$ und der Speicherbedarf bei $O(d)$. Ein Vorteil ist, dass nach jeder abgeschlossenen Iteration bereits ein gültiger Zug vorliegt. In Verbindung mit Zugsortierung können Ergebnisse früherer Iterationen genutzt werden, um vielversprechende Züge in tieferen Durchläufen bevorzugt zu untersuchen. Das Verfahren ist daher gut für zeitbeschränkte Suchprobleme geeignet. [5].

2.1.2.2 Transpositionstabelle

Transpositionstabellen (*engl. Transposition Tables*) sind ein gängiges Hilfsmittel in der Spielbaumsuche. Identische Spielstellungen können häufig über unterschiedliche Zugfolgen erreicht werden. Dadurch lassen sich bereits analysierte Positionen zwischenspeichern und ihre Bewertungen bei erneutem Auftreten wiederverwenden. Auf diese Weise kann der Rechenaufwand während der Suche reduziert werden. Der Einsatz von Transposition Tables erfordert zusätzlichen Speicher. Bei Speicherknappheit sind daher geeignete Ersetzungsstrategien notwendig.

Zur effizienten Nutzung von Transposition Tables wird häufig Zobrist-Hashing eingesetzt, ein speziell für Brettspiele entwickeltes Hashverfahren [13]. Die Stellung wird durch Zobrist-Hashing in einen 64-Bit-Hash überführt. Dazu werden zu Beginn mit einem festen Startwert des Zufallszahlengenerators (0xC0FFEE) deterministische Pseudozufallsschlüssel erzeugt (getrandbits(64)).

Das Spielfeld wird intern als lineares Array der Länge $17 \times 17 = 289$ repräsentiert. Für jeden Feldindex existiert eine kleine Tabelle, die jedem möglichen Feldzustand einen eigenen 64-Bit-Schlüssel

zuordnet. Die Zustände umfassen unter anderem die Besetzung mit Spieler 1 bzw. Spieler 2, ein freies Spielerfeld, ein freies Mauerfeld sowie ein belegtes Mauerfeld. Beim Berechnen des Hashwerts wird der Hashwert h initial auf 0 gesetzt. Anschließend wird für jedes Brettfeld der zum aktuellen Zustand passende Schlüssel mittels bitweisem XOR ($\oplus$) in h verknüpft.

Zusätzlich wird die Information, welcher Spieler am Zug ist, durch einen weiteren Schlüssel berücksichtigt. Ist Spieler 1 am Zug, wird ein fester Schlüssel per XOR verknüpft, bei Spieler 2 entfällt dieser Term. Auch die Anzahl der noch verfügbaren Mauern von Spieler 1 und Spieler 2 wird berücksichtigt, indem jeweils ein Schlüssel aus einer Tabellenzeile für die Werte $0, \ldots, 20$ per XOR eingebunden wird.

Der resultierende Wert h dient als schneller, annähernd eindeutiger Schlüssel für die Wiedererkennung von Stellungen in Transpositionstabellen und bei der Zugsortierung. Da Hashverfahren Kollisionen nicht vollständig ausschließen können, ist es grundsätzlich möglich, dass verschiedene Spielzustände denselben Hashwert erzeugen. Eine explizite Auflösung solcher Hashkollisionen erfolgt in dieser Arbeit nicht. Stattdessen wird die Kollisionswahrscheinlichkeit aufgrund der 64-Bit-Schlüsselbreite als vernachlässigbar klein angenommen.

Ein Eintrag in der Transposition Table besteht typischerweise aus folgenden Komponenten[3] [1]:

- **hash:** Ein mittels Zobrist-Hashing erzeugter Hashwert der aktuellen Spielstellung, der als Schlüssel für den Tabelleneintrag dient.
- **move:** Der beste bekannte Zug aus der entsprechenden Spielstellung.
- **score:** Die Bewertung der Spielstellung, entweder als exakter Wert oder als obere bzw. untere Schranke.
- **flag:** Kennzeichnet die Art der gespeicherten Bewertung:
  - **EXACT:** Exakter Bewertungswert, falls $\alpha < \text{score} < \beta$.
  - **LOWERBOUND:** Untere Schranke, falls $\beta \leq \text{score}$.
  - **UPPERBOUND:** Obere Schranke, falls $\text{score} \leq \alpha$.
- **depth:** Die relative Suchtiefe, für die der Eintrag gültig ist. Sie ergibt sich aus der maximalen Suchtiefe abzüglich der bereits ausgeführten Züge und erlaubt den Vergleich unterschiedlich tief berechneter Einträge.

Bevor ein Knoten weiter untersucht wird, prüft der Algorithmus, ob die zugehörige Spielstellung bereits in der Transposition Table gespeichert ist. Ist dies der Fall und ist die gespeicherte Tiefe größer oder gleich der verbleibenden Suchtiefe, hängt das weitere Vorgehen vom gespeicherten *flag* ab.

Bei *flag* = EXACT wird die gespeicherte Bewertung direkt zurückgegeben. Bei *flag* = LOWERBOUND wird α auf $\max(\alpha, \text{score})$ aktualisiert und *flag* = UPPERBOUND entsprechend β auf $\min(\beta, \text{score})$. Gilt anschließend $\beta \leq \alpha$, kann ein Alpha-Beta-Cutoff durchgeführt werden [1].

[3] https://www.chessprogramming.org/Transposition_Table, Zugriff: 21.02.2026

2.1.2.3 Zugsortierung

Eine effektive Zugsortierung erhöht die Wahrscheinlichkeit früher Alpha-Beta-Abschneidungen. In der implementierten Variante kommen drei Mechanismen zum Einsatz.

Als Grundordnung dient eine statische Heuristik (*Baseline-Sortierung*). Sie reiht Figurenzüge vor Wandzügen ein und sortiert die Figurenzüge zusätzlich nach Richtungspriorität. Wandzüge werden nach der Manhattan-Distanz zur aktuellen Position und der verbleibenden Zieldistanz geordnet.[4] Der Gedanke hinter dieser Reihenfolge ist, dass pro Stellung in der Regel weniger legale Figurenzüge als legale Wandzüge vorhanden sind. Werden zunächst die Figurenzüge untersucht, könnten theoretisch frühe Alpha-Beta-Cutoffs dazu führen, dass ein Teil der Wandnachfolger nicht mehr expandiert werden muss. Dies ist jedoch keine Aussage darüber, dass Wandzüge grundsätzlich weniger vorteilhaft wären. Vielmehr handelt es sich um eine heuristische Ordnungsentscheidung zur Reduktion des Suchaufwands. Wandzüge bleiben als legale Kandidaten im Suchraum grundsätzlich erhalten. Sie werden jedoch wie alle anderen Züge in Abhängigkeit von der Zugreihenfolge und den resultierenden Alpha-Beta-Cutoffs nicht in jedem Knoten vollständig expandiert. In konkreten Stellungen können sie weiterhin strategisch besonders vorteilhaft sein.

Innerhalb der iterativen Tiefensuche werden nach jeder vollständig abgeschlossenen Suchtiefe für die direkten Nachfolger der Wurzelstellung Bewertungswerte mit der Zustandsheuristik berechnet. Diese Werte werden in einer Zuordnung gespeichert und für eine heuristikbasierte Zugreihenfolge genutzt (*Heuristikbasierte Sortierung*). Ab der zweiten Suchtiefe dienen diese Werte zur Festlegung der Zugreihenfolge auf der Wurzelebene, sodass vielversprechende Züge dort bevorzugt zuerst untersucht werden.

Sofern in der Transpositionstabelle ein gespeicherter bester Zug für die aktuelle Stellung existiert, wird das zugehörige Kind zunächst an den Anfang der Zugliste gesetzt. Dies geschieht vor der heuristischen bzw. Baseline-Sortierung und wird im Folgenden als *TT-Zugpriorität* bezeichnet. Die unmittelbar folgende Sortierung kann diese Reihenfolge jedoch wieder verändern. Ein fester Vorrang des TT-Zugs gegenüber den übrigen Ordnungsmechanismen besteht damit nicht. Grundsätzlich könnte die Transpositionstabelle auch für eine weitergehende Sortierung aller legalen Züge genutzt werden, indem für jede Kindstellung ein TT-Eintrag abgefragt und der gespeicherte Wert als Sortierschlüssel herangezogen wird. In der vorliegenden Implementierung erfolgt jedoch kein vollständiger Vergleich aller Kindstellungen auf Basis ihrer TT-Bewertungen. Ein Grund dafür ist, dass für viele Kindstellungen insbesondere in frühen Suchphasen kein verwertbarer TT-Eintrag vorliegt. Zudem enthalten TT-Einträge häufig nur Schranken statt direkt vergleichbarer exakter Bewertungen, und Einträge aus geringer gespeicherter Tiefe können für die aktuelle Suche nur eingeschränkt aussagekräftig sein. Die Transpositionstabelle ergänzt die Zugsortierung somit, ersetzt jedoch weder die heuristische noch die Baseline-Sortierung.

[4] https://github.com/pavlosdais/Quoridor, Zugriff: 21.02.2026

Algorithm 2 Move Ordering

```
if TT[Zobrist(game_state)].best_move exists then
    Move the matching child to the front of children
end if
if move_ordering_map exists then
    if maximizing_player then
        Sort children descending by move_ordering_map[hash(child)]; unknown keys ← −∞
    else
        Sort children ascending by move_ordering_map[hash(child)]; unknown keys ← +∞
    end if
else
    Split children into pawn_moves and wall_moves
    Sort pawn_moves ascending by PawnPriority(move, game_state)
    Sort wall_moves ascending by WallDistance(move, game_state)
    children ← pawn_moves + wall_moves
end if
                                ▷ Heuristic sort can reorder children; TT move is not necessarily first.
return children
```

2.2 Monte Carlo Tree Search

Zur Realisierung eines spielstarken Agenten für Quoridor wird in dieser Arbeit MCTS eingesetzt. Klassische Suchverfahren stoßen aufgrund des hohen Verzweigungsfaktors und der kombinatorischen Komplexität des Spiels schnell an ihre Grenzen. Eine vollständige Durchsuchung des Spielbaums ist daher nicht praktikabel. MCTS begegnet diesem Problem durch stochastische Stichprobenziehung und die schrittweise verstärkte Untersuchung vielversprechender Spielzweige. Der gesamte Suchraum muss dabei nicht systematisch expandiert werden.

2.2.1 Basis-Algorithmus

Der entwickelte MCTS-Agent folgt dem klassischen vierphasigen Ablauf aus Selection, Expansion, Simulation und Backpropagation (vgl. Abb. 2.2). Dieser Zyklus wird iterativ wiederholt, bis ein vorgegebenes Rechenbudget ausgeschöpft ist. Das Budget wird entweder durch ein Zeitlimit oder durch eine maximale Anzahl an Simulationen bestimmt.[5] [2].

[5] https://www.geeksforgeeks.org/machine-learning/monte-carlo-tree-search-mcts-in-machine-learning/, Zugriff: 21.02.2026

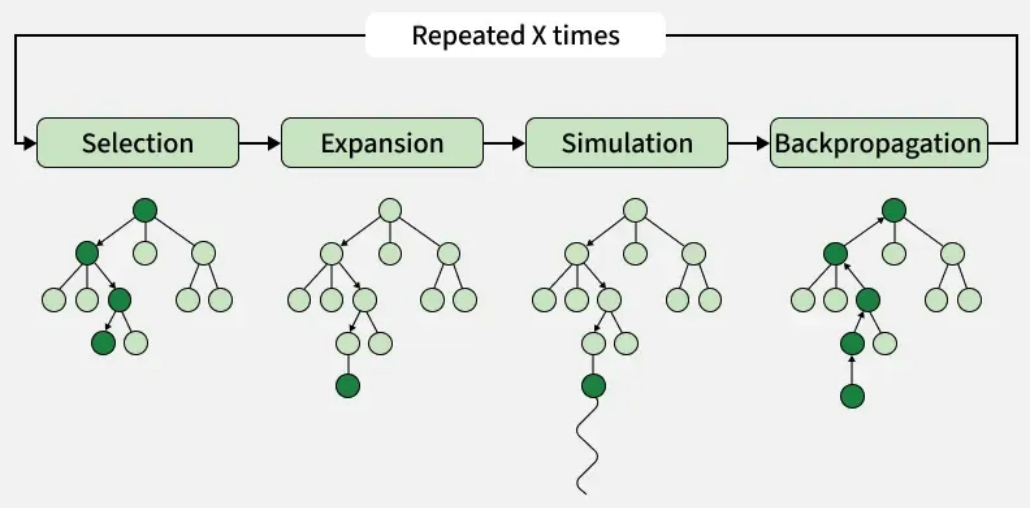


Abbildung 2.2: MCTS-Schritte [6]

Selection

In der Selektionsphase wird ausgehend vom Wurzelknoten schrittweise jeweils ein Kindknoten ausgewählt, bis ein Blattknoten erreicht ist. Zur Balance zwischen Exploitation bereits erfolgreicher Züge und Exploration seltener besuchter Knoten wird die UCT-Formel (*Upper Confidence Bound applied to Trees*) [4] verwendet. Sie kombiniert die bisherige Gewinnrate eines Knotens mit einem explorativen Term.

Die finale Zugauswahl am Wurzelknoten erfolgt jedoch nicht anhand des höchsten UCT-Wertes, sondern über das Robust-Child-Kriterium. Es wird der Kindknoten mit der höchsten Besuchszahl gewählt. Dies reduziert die Anfälligkeit gegenüber statistischen Ausreißern und erhöht die Stabilität der Entscheidung.

Expansion

Sobald ein Blattknoten erreicht wird, werden in der Expansionsphase dessen legale Folgezustände als neue Kindknoten zum Suchbaum hinzugefügt. Der Suchbaum wächst damit schrittweise in jene Bereiche, die durch die Selektionsstrategie als vielversprechend identifiziert wurden.

Simulation (Rollout)

Nach der Expansion wird vom neu erzeugten Knoten eine Simulation bis zum Spielende durchgeführt. In der klassischen Variante erfolgt diese zufällig und liefert ein Endergebnis (Sieg oder Niederlage), das als Grundlage für die statistische Bewertung des Knotens dient.

[6] https://www.geeksforgeeks.org/machine-learning/monte-carlo-tree-search-mcts-in-machine-learning/, Zugriff: 23.03.2026

Backpropagation

In der Backpropagationsphase wird das Simulationsergebnis vom Blattknoten bis zum Wurzelknoten zurückpropagiert. Für jeden besuchten Knoten werden dabei der Besuchszähler und die Gewinnstatistik aktualisiert. Bei Zwei-Personen-Nullsummenspielen wird das Ergebnis dabei alternierend mit wechselndem Vorzeichen an die Elternknoten weitergegeben, da ein positiver Ausgang für einen Spieler aus Sicht des Gegners negativ zu bewerten ist. Mit steigender Simulationsanzahl stabilisiert sich die Bewertung, weil zufällige Schwankungen zunehmend ausgeglichen werden.

2.2.2 Upper Confidence Bound applied to Trees (UCT)

Ein zentrales Element der Selektionsphase ist die UCT-Strategie (Upper Confidence Bound applied to Trees), die eine kontrollierte Balance zwischen der Ausnutzung vielversprechender Züge (Exploitation) und der Erkundung wenig besuchter Alternativen (Exploration) gewährleistet [4].

Die Auswahl eines Kindknotens erfolgt anhand folgender Bewertungsfunktion:

$$UCT_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N)}{n_i}} \tag{2.2}$$

Dabei bezeichnet:

- w_i die Anzahl der gewonnenen Simulationen des Knotens i,
- n_i die Anzahl der Besuche dieses Knotens,
- N die Anzahl der Besuche des Elternknotens,
- c einen explorativen Konstantenfaktor.

Der erste Term $\frac{w_i}{n_i}$ repräsentiert die empirische Gewinnrate des Knotens und fördert die Auswahl erfolgreicher Züge. Der zweite Term stellt den explorativen Anteil dar und bevorzugt selten besuchte Knoten. Mit zunehmender Besuchszahl n_i nimmt der Einfluss des Explorationsterms ab.

Der Parameter c steuert die Gewichtung zwischen Exploration und Exploitation. Größere Werte fördern die Erkundung selten besuchter Knoten, während kleinere Werte die Suche auf bereits gut bewertete Züge fokussieren. Da der optimale Wert spielabhängig ist, wird c in der Praxis experimentell bestimmt.

Zur Parametrierung von c kann ein systematisches Grid-Search-Vorgehen[7] eingesetzt werden. Mehrere MCTS-Instanzen mit unterschiedlichen c-Werten (z. B. $c \in \{1.0, 1.5, 2.0, 2.5, \dots\}$) treten in einer großen Anzahl von Partien gegeneinander an. Optional können zusätzlich Spiele gegen eine externe Referenz-KI (z. B. eine Minimax-/Alpha-Beta-Variante) durchgeführt werden, um die Robustheit der Wahl zu prüfen. Als Auswahlkriterium dient die erzielte Gewinnrate bzw. ein aggregiertes Performanzmaß über alle Partien. In der Literatur werden für UCT in vielen Szenarien Werte im Bereich um

[7] https://web.informatik.uni-mannheim.de/cmeilicke/grundlagen/KIX-06-MCTS.pdf,Zugriff: 21.02.2026

$c = \sqrt{2}$ als praktikabler Ausgangspunkt genannt, wobei die endgültige Wahl durch die beschriebenen Experimente abgesichert werden sollte (vgl. Kocsis und Szepesvári (2006)) [4].

2.2.3 Strategien zur Effizienzsteigerung des MCTS

Ähnlich wie bei der Alpha-Beta-Suche existieren auch für MCTS verschiedene Erweiterungsstrategien zur weiteren Steigerung der Leistungsfähigkeit.

2.2.3.1 Lazy Expansion

Bei der Expansion eines Knotens müssen alle legalen Züge der zugehörigen Spielstellung berechnet werden. In Quoridor ist dies besonders aufwendig. Für jeden potenziellen Wandzug muss per Astar geprüft werden, ob beide Spieler noch einen Weg zum Ziel besitzen.

Um den Aufwand zu minimieren, wird Lazy Expansion eingesetzt. Die Liste der unerforschten Aktionen wird nicht bereits bei der Knotenerstellung berechnet, sondern erst beim ersten Expansionsaufruf initialisiert. Bei jedem weiteren Besuch wird lediglich ein weiterer Zug entnommen, bis alle Nachfolger expandiert wurden.

Der Vorteil besteht darin, dass in einem typischen MCTS-Durchlauf die meisten Knoten zwar durchlaufen, aber nicht expandiert werden. Ohne Lazy Expansion müsste dennoch für jeden dieser Knoten die vollständige Zugberechnung ausgeführt werden. Insbesondere bei hoher Simulationsanzahl würde dies zu erheblichem Mehraufwand führen.

Algorithm 3 MCTS mit Lazy Expansion

Require: root_state, N (number of simulations), C (exploration constant)
Ensure: action with maximum visit count at root

```
root ← NODE(root_state)
for i ← 1 to N do
    leaf ← SELECT(root, C)                          ▷ returns a terminal or not fully expanded node
    if not ISTERMINAL(leaf.state) then
        if leaf.untried_actions = null then          ▷ Lazy: initialize only on first expansion
            leaf.untried_actions ← GETLEGALACTIONS(leaf.state)
        end if
        if leaf.untried_actions ≠ ∅ then
            child ← EXPANDONE(leaf)
            result ← SIMULATE(child.state)
            BACKPROPAGATE(child, result)
        else                                         ▷ No legal moves available
            result ← SIMULATE(leaf.state)
            BACKPROPAGATE(leaf, result)
        end if
    else                                             ▷ Terminal node: winner is evaluated directly
        result ← TERMINALRESULT(leaf.state)
        BACKPROPAGATE(leaf, result)
    end if
end for
return ROBUSTCHILD(root)                             ▷ child with highest visit count
```

2.2.3.2 Heuristikbasierte Bewertung

Als erste Auswertungsstrategie wird eine heuristische Stellungsbewertung verwendet. Dabei wird keine vollständige Zugsequenz bis zum Spielende generiert. Stattdessen wird die aktuelle Stellung unmittelbar durch eine Bewertungsfunktion analysiert. Die verwendete Heuristik entspricht der in Abschnitt 2.1.1 beschriebenen Bewertungsfunktion und setzt sich aus denselben vier gewichteten Komponenten zusammen.

Der resultierende Score wird auf $[-20, 20]$ begrenzt und mittels Tangens hyperbolicus in eine Gewinnwahrscheinlichkeit $p \in [0, 1]$ überführt. Bei $p > 0{,}7$ wird das Ergebnis als Gewinn gewertet, bei $p < 0{,}3$ als Niederlage. In ausgeglichenen Stellungen wird eine geringe Zufallskomponente ($\pm 5\,\%$) hinzugefügt, um exploratives Verhalten zu fördern. Bereits berechnete Stellungen werden über eine Transpositionstabelle (Zobrist-Hash) gespeichert, sodass identische Zustände nicht mehrfach ausgewertet werden müssen.

Die Rückgabe an MCTS liegt damit praktisch in $\{-1, +1\}$, obwohl die zugrunde liegende Heuristik kontinuierliche Information enthält. Diese Diskretisierung ist keine notwendige Eigenschaft von MCTS, sondern eine Modellierungsentscheidung. Binäre Rückmeldungen sind einfach zu interpretieren und passen gut zur klassischen UCT-Logik, führen jedoch dazu, dass Informationen über annähernd ausgeglichene Stellungen verloren gehen. Grundsätzlich wäre daher auch ein zusätzlicher

Neutralwert 0 denkbar, um Stellungen im Bereich neutraler Heuristikwerte als ausgeglichen zu behandeln.

Darüber hinaus ist MCTS nicht auf binäre Belohnungen beschränkt. UCT kann auch auf kontinuierliche Rückgabewerte in $[-1, 1]$ angewendet werden, die entlang des Suchpfads gemittelt werden. Die in dieser Arbeit gewählte Diskretisierung auf $\{-1, +1\}$ ist daher als pragmatische Vereinfachung zu verstehen.

Algorithm 4 Heuristic-Based Simulation

Require: state, transposition_table
Ensure: result $\in \{-1, 1\}$
original_player $\leftarrow$ state.player_one
state_hash $\leftarrow$ COMPUTEZOBRISTHASH(state)
if state_hash $\in$ transposition_table **then**
    return transposition_table[state_hash]
end if
if ISTERMINAL(state) **then**
    result $\leftarrow +1$ if original_player is winner, else -1
    transposition_table[state_hash] $\leftarrow$ result
    return result
end if
score $\leftarrow$ STATEEVALUATIONHEURISTIC(state, original_player)
score $\leftarrow$ CLAMP(score, -20, $+20$)
win_prob $\leftarrow \tanh\left(\frac{\text{score}}{5.0}\right)$
win_chance $\leftarrow \frac{\text{win\_prob} + 1}{2}$
if win_chance > 0.7 **then**
    result $\leftarrow +1$
else if win_chance < 0.3 **then**
    result $\leftarrow -1$
else
    noise $\leftarrow \mathcal{U}(-0.05,\ 0.05)$
    adjusted_chance $\leftarrow$ CLAMP(win_chance + noise, 0, 1)
    result $\leftarrow +1$ with probability adjusted_chance, else -1
end if
transposition_table[state_hash] $\leftarrow$ result
return result

2.2.3.3 Playout-Policy mit strategischer Gewichtung

Neben der heuristikbasierten Bewertung wurde auch eine strukturierte Playout-Policy implementiert. Sie verwendet anstelle zufälliger Züge eine regelbasierte Zugauswahl [10]. Da rein zufällige Rollouts in Quoridor häufig unrealistische Spielverläufe erzeugen, liefern sie oft wenig stabile Bewertungsschätzungen. Dies kann die Qualität der Zugauswahl beeinträchtigen. Zur Lösung dieses Problems

folgt die Policy einer hierarchischen Zugauswahl mit drei Verhaltensmustern.[8].

Mit einer Wahrscheinlichkeit von $70\,\%$ wird ein Zug gewählt, der den eigenen Spielstein entlang des aktuell berechneten BFS-Kürzestpfades in Richtung Ziel bewegt. Die dazu benötigte Pfadstruktur wird zwischengespeichert und bei Änderungen der Stellung, etwa durch Figurenbewegungen oder Mauerplatzierungen, neu berechnet. Dadurch werden innerhalb unveränderter Stellungen wiederholte BFS-Aufrufe vermieden, während zugleich ein zielgerichtetes Spielverhalten unterstützt wird.

Mit der verbleibenden Wahrscheinlichkeit von $30\,\%$ wird, sofern möglich, ein Wandzug gewählt, falls noch Mauern verfügbar sind. Als wahrscheinlich relevante Wandpositionen gelten dabei alle Platzierungen, deren Manhattan-Distanz zu einer der beiden Spielfiguren höchstens 5 beträgt. Sind keine solchen Positionen vorhanden, wird aus der Menge aller legalen Wandzüge gleichverteilt gewählt.

Wenn weder der gezielte Vorwärtszug auf dem kürzesten Weg noch ein Wandzug gewählt wird, greift die Policy auf alternative Figurenzüge zurück. Dazu zählen Züge entlang des kürzesten Weg in entgegengesetzter Richtung oder Züge, die die BFS-Distanz zur Zielreihe eher verlängern. Ihr Zweck besteht darin, die Rollouts weniger deterministisch zu machen. Würde die Simulation ausschließlich dem jeweils kürzesten Weg zum Ziel folgen, würden ähnliche Spielverläufe entstehen, sodass viele Rollouts nur wenig zusätzliche Information liefern. Durch die Einbeziehung alternativer, auch weniger zielgerichteter Zugfolgen wird die Vielfalt der simulierten Partien erhöht. Dadurch kann MCTS unterschiedliche Entwicklungsmöglichkeiten der Stellung breiter erfassen, anstatt in den Simulationen immer wieder ähnliche Abläufe zu reproduzieren.

Wird innerhalb der maximalen Simulationstiefe von 100 Zügen kein Terminalzustand erreicht, wird die Simulation abgebrochen. Anschließend erfolgt die Bewertung der Stellung mithilfe derselben heuristischen Bewertungsfunktion wie bei der heuristikbasierten Simulation. Gegenüber rein zufälligen Rollouts reduziert diese strukturierte Playout-Policy die Varianz der Simulationsergebnisse und ermöglicht stabilere Schätzungen der Erfolgswahrscheinlichkeit. Dem steht jedoch ein erhöhter Rechenaufwand gegenüber, da wiederholt BFS-Berechnungen erforderlich sind.

[8] https://github.com/gorisanson/quoridor-ai, Zugriff: 21.02.2026

Algorithm 5 Playout-Policy Simulation

Require: state, max_depth, transposition_table
Ensure: result $\in \{-1, +1\}$

```
original_player ← state.player_one
state_hash ← COMPUTEZOBRISTHASH(state)
if state_hash ∈ transposition_table then
    return transposition_table[state_hash]
end if
if ISTERMINAL(state) then
    result ← +1 if original_player is winner, else −1
    transposition_table[state_hash] ← result
    return result
end if
current ← COPY(state)
move_count ← 0
while not ISTERMINAL(current) and move_count < max_depth do
    r ← RANDOM([0, 1))
    if r < 0.7 then                                  ▷ 70%: move along shortest path to goal
        next_pos ← NEXTONSHORTESTPATH(current)
        MOVEPAWN(current, next_pos)
    else if current player has walls remaining then   ▷ 30%: place a probable wall near pawns
        wall ← CHOOSEPROBABLEWALL(current)
        if wall ≠ null then
            PLACEWALL(current, wall)
        else                                          ▷ Fallback: move backwards (longest path)
            prev_pos ← PREVONLONGESTPATH(current)
            MOVEPAWN(current, prev_pos)
        end if
    else                                              ▷ No walls left: move backwards
        prev_pos ← PREVONLONGESTPATH(current)
        MOVEPAWN(current, prev_pos)
    end if
    move_count ← move_count +1
end while
if ISTERMINAL(current) then
    result ← +1 if original_player is winner, else −1
else                                                  ▷ Depth limit reached: fall back to heuristic evaluation
    result ← HEURISTICSIMULATION(current, original_player)
end if
transposition_table[state_hash] ← result
return result
```

3 Ergebnisse

Die Implementierung der Suchalgorithmen (Alpha-Beta-Suche und Monte Carlo Tree Search) wurde in Python vorgenommen. NumPy kam für numerische Operationen und die Verwaltung der Spielfeldrepräsentation zum Einsatz. Die Auswertung der Experimente sowie die grafische Darstellung der Ergebnisse erfolgten mit Matplotlib.

Sämtliche Experimente wurden auf einem einheitlichen Testsystem mit folgender Konfiguration durchgeführt:

- **Prozessor:** Apple M2
- **Arbeitsspeicher:** 8 GB RAM
- **Betriebssystem:** macOS
- **Programmiersprache und Laufzeitumgebung:** Python 3.12.4

3.1 Alpha-Beta

3.1.1 Test 1: Laufzeitvergleich der Alpha-Beta-Varianten

Im ersten Experiment wurde die Berechnungszeit verschiedener Alpha-Beta-Varianten verglichen (vgl. Abb. 3.1). Ziel ist es, zu analysieren, inwiefern die jeweiligen Erweiterungen die Suchgeschwindigkeit beeinflussen.

Verglichen wurden die folgenden vier Varianten:

- **AB**: Standard-Alpha-Beta-Suche ohne zusätzliche Optimierungen
- **AB+TT**: Alpha-Beta-Suche mit Transpositionstabelle
- **AB+ID+MO**: Alpha-Beta-Suche mit iterativer Tiefensuche und Zugsortierung
- **AB+ID+MO+TT**: Alpha-Beta-Suche mit iterativer Tiefensuche, Zugsortierung und Transpositionstabelle

In der Variante AB+ID+MO wird bewusst keine Transpositionstabelle verwendet. Den Effekt der iterativen Tiefensuche in Kombination mit der Zugsortierung soll isolierter betrachtet werden. In der Literatur wird iterative Tiefensuche häufig gemeinsam mit Transpositionstabellen eingesetzt, um Mehrfachberechnungen zu vermeiden. Diese Trennung ist bei der Interpretation der Ergebnisse zu berücksichtigen.

Für die Experimente wurden insgesamt 10 fest definierte, unterschiedliche Spielpositionen verwendet. Jede Variante wurde für die Suchtiefen 4, 5 und 6 ausgeführt.

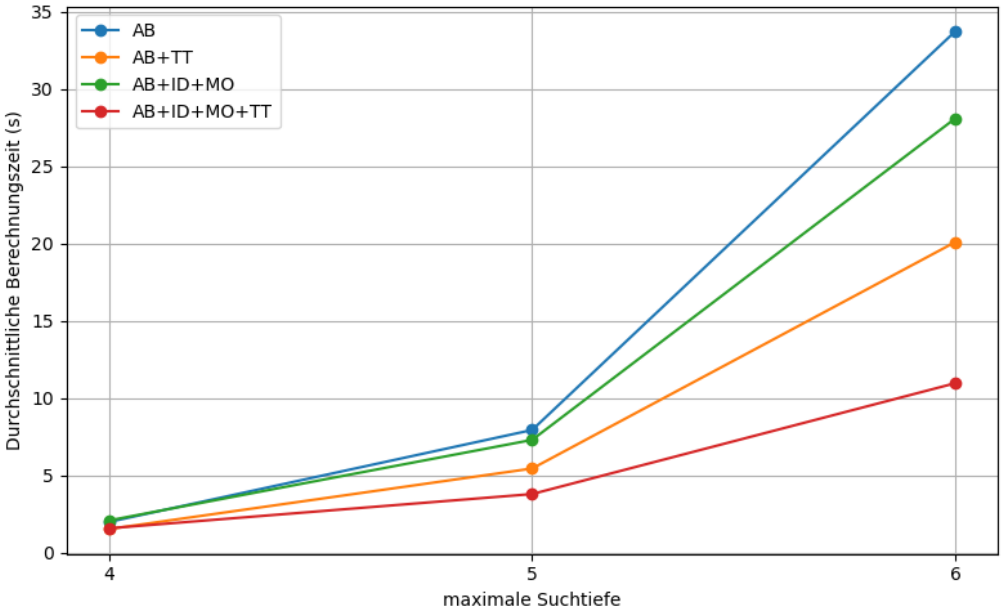


Abbildung 3.1: Durchschnittliche Berechnungszeit der Alpha-Beta-Varianten

Zusätzlich wurde die Berechnungszeit jeder Variante relativ zur Basisvariante AB normiert, indem deren Laufzeit für jede Suchtiefe auf $100\,\%$ gesetzt wurde.

Suchtiefe	AB	AB+TT	AB+ID+MO	AB+ID+MO+TT
4	100.0 %	76.8 %	105.0 %	79.2 %
5	100.0 %	68.5 %	91.9 %	47.7 %
6	100.0 %	59.6 %	83.3 %	32.5 %

Tabelle 3.1: Relative durchschnittliche Berechnungszeit der Alpha-Beta-Varianten

Die Ergebnisse zeigen, dass alle Erweiterungen gegenüber der Basisvariante AB zu einer reduzierten Berechnungszeit führen. Die vollständig erweiterte Variante AB+ID+MO+TT erzielt dabei die stärkste Reduktion. Bei Suchtiefe 4 beträgt die Laufzeit noch $79{,}2\,\%$ der Basisvariante, bei Suchtiefe 5 bereits $47{,}7\,\%$ und bei Suchtiefe 6 lediglich $32{,}5\,\%$. Die Variante AB+TT reduziert die Laufzeit ebenfalls, während AB+ID+MO den geringsten Effekt aufweist. Zudem ist erkennbar, dass der Effizienzgewinn der Erweiterungen mit zunehmender Suchtiefe zunimmt.

3.1.2 Test 2: Maximal erreichte Suchtiefe unter festen Zeitlimits

Im zweiten Experiment wurde untersucht, welche maximale Suchtiefe die einzelnen Alpha-Beta-Varianten innerhalb eines vorgegebenen Zeitlimits erreichen. Um verschiedene Spielphasen abzudecken, wurden ausgehend von der Startposition Stellungen nach 10, 20, 30, 40, 50 und 60 Zufallszügen

erzeugt. Diese Stellungen repräsentieren näherungsweise Eröffnung, Mittelspiel und Endspiel. Pro Phase wurden drei reproduzierbare Positionen generiert, sodass insgesamt 18 Testpositionen vorliegen. Jede Variante wurde auf jeder Position mit Zeitlimits von 1 bis 10 Sekunden getestet.

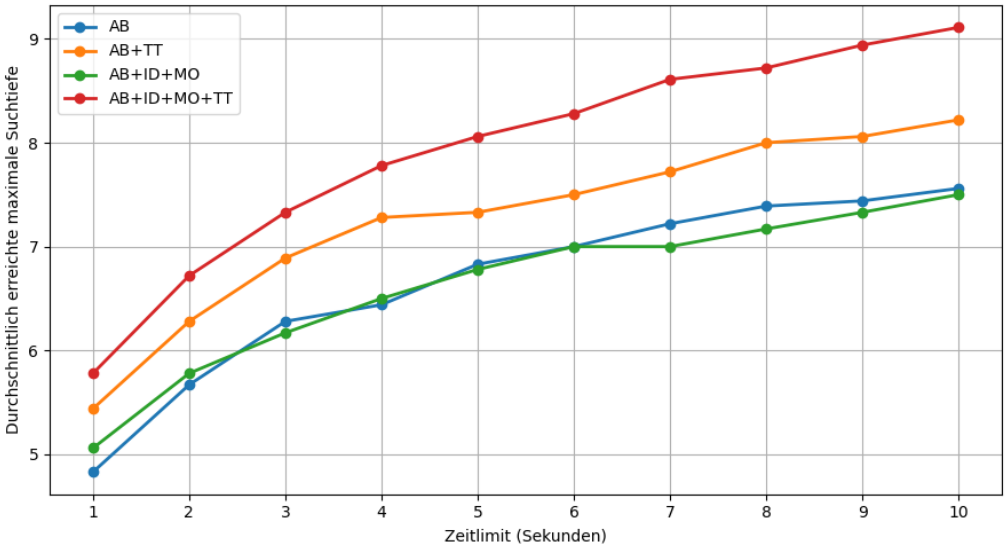


Abbildung 3.2: Durchschnittlich erreichte maximale Suchtiefe

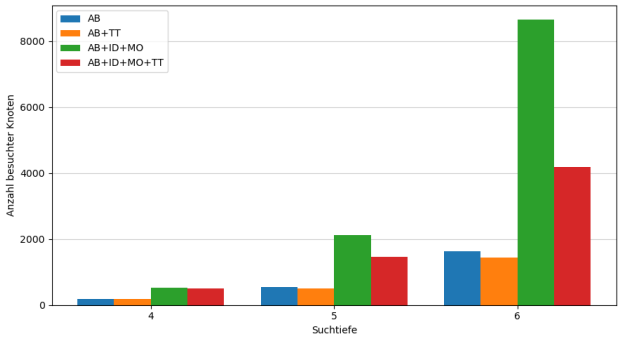
AB+ID+MO+TT erreicht über alle Zeitlimits hinweg die höchste durchschnittliche Suchtiefe. Bei einem Zeitlimit von 1 Sekunde beträgt die mittlere Suchtiefe 5,78 und steigt bei 10 Sekunden auf 9,11 an. AB+TT belegt durchgehend den zweiten Platz mit einer mittleren Suchtiefe von 5,44 bei 1 Sekunde und 8,22 bei 10 Sekunden. Die Basisvariante AB und AB+ID+MO verlaufen über weite Strecken nahezu parallel und liegen häufig nah beieinander. AB startet bei 4,83 und erreicht 7,56, AB+ID+MO beginnt bei 5,06 und endet bei 7,50. Bei niedrigen Zeitlimits liegt AB+ID+MO leicht vorne, während AB ab mittleren Zeitlimits aufschließt und die beiden Kurven sich teilweise überschneiden. Mit zunehmendem Zeitlimit steigt die erreichbare Suchtiefe aller Varianten an, wobei der Zuwachs bei höheren Zeitlimits geringer ausfällt.

3.1.3 Test 3: Vergleich der Anzahl besuchter Knoten

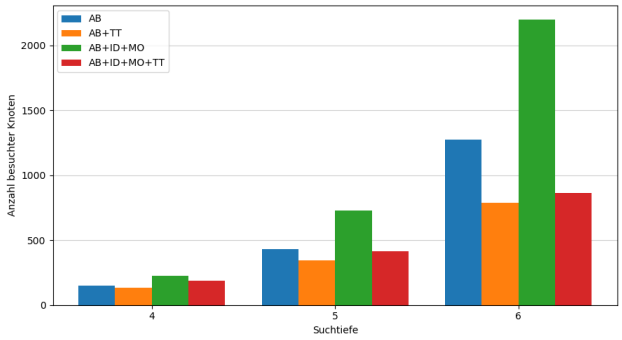
Im dritten Experiment wurde die Anzahl der besuchten Knoten verschiedener Alpha-Beta-Varianten miteinander verglichen. Um unterschiedliche Spielphasen abzubilden, werden drei fest definierte Spielstellungen betrachtet, nämlich die Startposition, eine Mittelspielstellung sowie eine Endsituation.

Alle Varianten werden auf jeder der drei Positionen mit den Suchtiefen 4, 5 und 6 ausgeführt. Bei den Varianten AB und AB+TT entspricht die gemessene Knotenzahl der Anzahl der besuchten Knoten einer einzelnen Suche. Diese Suche reicht jeweils bis zur entsprechenden Zieltiefe. Bei den Varianten AB+ID+MO und AB+ID+MO+TT umfasst die Knotenzahl hingegen alle Knoten, die während der iterativen Tiefensuche besucht wurden. Dies schließt alle Suchtiefen von 1 bis zur jeweiligen Endtiefe

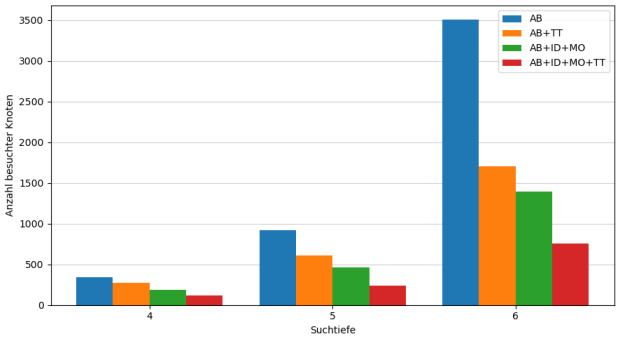
ein. Für jede dieser Stellungen wird jeweils eine einzelne Suche von der Wurzel aus durchgeführt, vollständige Partien werden in diesem Experiment nicht simuliert.



(a) Startposition



(b) Mittelspielstellung



(c) Endsituation

Abbildung 3.3: Vergleich der Anzahl besuchter Knoten für Alpha-Beta-Varianten in drei fest definierten Brettstellungen (Start-, Mittel- und Endstellung). Für jede Stellung wird jeweils eine einzelne Suche von der Wurzel aus mit den Suchtiefen 4, 5 und 6 durchgeführt. Gemessen wird die Anzahl der während dieser Suche besuchten Knoten. Es werden keine vollständigen Partien simuliert.

Die Wirkung der untersuchten Erweiterungen unterscheidet sich zwischen den Spielphasen. Am klarsten lässt sich der Effekt der Transpositionstabelle beurteilen, da mit AB+TT sowie AB+ID+MO+TT jeweils direkte Vergleichsvarianten vorliegen. In allen drei Spielphasen reduziert TT die Anzahl besuchter Knoten gegenüber der jeweiligen Variante ohne TT. In der Startposition sinkt die Knotenzahl bei Suchtiefe 6 von 1.631 auf 1.438 für AB gegenüber AB+TT sowie von 8.644 auf 4.186 für AB+ID+MO gegenüber AB+ID+MO+TT. Im Mittelspiel zeigt sich derselbe Trend mit einer Reduktion von 1.248 auf 762 bzw. von 2.069 auf 841. Besonders ausgeprägt ist die Wirkung in der Endsituation, in der AB+TT die Knotenzahl von 3.505 auf 1.706 senkt und AB+ID+MO+TT gegenüber AB+ID+MO von 1.394 auf 754 reduziert.

Weniger eindeutig isolierbar ist der spezifische Beitrag von iterativer Tiefensuche und Zugsortierung, da beide nur in Kombination untersucht wurden. Betrachtet man AB+ID+MO im Vergleich zu AB, zeigt sich ein spielphasenabhängiges Muster. In der Startposition liegt die Knotenzahl bei Suchtiefe 6 mit 8.644 deutlich über dem Wert der Basisvariante AB mit 1.631, im Mittelspiel mit 2.069 ebenfalls

noch darüber. In der Endsituation kehrt sich dieses Verhältnis jedoch um. Dort reduziert AB+ID+MO die Knotenzahl von 3.505 auf 1.394.

Insgesamt zeigt sich, dass TT in allen getesteten Phasen konsistent knotensenkend wirkt, während die Kombination aus ID und MO von der Struktur der jeweiligen Stellung abhängt. Eine vollständige Trennung der Einzeleffekte von ID und MO ist auf Grundlage der vorliegenden Varianten allerdings nicht möglich.

3.1.4 Test 4: Spielstärke der Alpha-Beta-Variante gegen Referenzgegner

Zur Bewertung der praktischen Spielstärke wurde AB+ID+MO+TT gegen zwei Referenzgegner untersucht, einen Zufallsspieler und einen heuristischen Greedy-Spieler.

Der Zufallsspieler wählt jeden Zug zufällig ohne strategische Bewertung. Pro Suchtiefe von 1 bis 4 wurden 20 Partien gespielt. Der Algorithmus spielte dabei jeweils 10 Partien als Spieler 1 und 10 als Spieler 2. Im Folgenden bezeichnet das Zuglimit eine feste Obergrenze für die Anzahl zulässiger Züge pro Partie (bzw. Halbzüge, da jede Spieleraktion als Zug gezählt wird). Es dient dazu, sehr lange oder nicht terminierende Partien, etwa durch Endlossituationen, zuverlässig abzubrechen. Wird diese Grenze erreicht, ohne dass ein regulärer Sieg festgestellt wurde, wird die Partie als unentschieden gewertet. Es galt ein Zuglimit von 300 Zügen.

Der Greedy-Spieler bewegt seine Figur entlang des kürzesten BFS-Pfades, sofern er über gleich viele oder mehr Wände als der Gegner verfügt. Andernfalls setzt er eine den Gegner verlangsamende Wand. Pro Suchtiefe wurden 60 Partien auf 30 zufälligen Startpositionen gespielt, mit einem Zuglimit von 200 Zügen.

Tabellen 3.2 und 3.3 zeigen für jede untersuchte Suchtiefe die Ergebnisse getrennt nach Spielerrolle. Angegeben werden jeweils die Anzahl der Siege von AB+ID+MO+TT (S), die Anzahl der Unentschieden (U) sowie die Anzahl der vom Referenzgegner gewonnenen Partien (R). Die Ergebnisse beider Spielerrollen werden zudem zusammengefasst und als Gesamtwert ausgewiesen. Da ein Teil der Partien unentschieden endet, werden Unentschieden bei der Berechnung der Siegrate nicht berücksichtigt. Die Siegrate wird daher ausschließlich über die entschiedenen Partien berechnet:

$$\mathrm{SR} = \frac{S}{S + R} \tag{3.1}$$

Tiefe	P1 (S/U/R)	P1 SR	P2 (S/U/R)	P2 SR	Ges. (S/U/R)	Ges. SR
1	20 / 0 / 0	100.0%	20 / 0 / 0	100.0%	40 / 0 / 0	100.0%
2	20 / 0 / 0	100.0%	20 / 0 / 0	100.0%	40 / 0 / 0	100.0%
3	20 / 0 / 0	100.0%	20 / 0 / 0	100.0%	40 / 0 / 0	100.0%
4	20 / 0 / 0	100.0%	20 / 0 / 0	100.0%	40 / 0 / 0	100.0%

Tabelle 3.2: Siegrate von AB+ID+MO+TT gegen einen Zufallsspieler

Tiefe	P1 (S/U/R)	P1 SR	P2 (S/U/R)	P2 SR	Ges. (S/U/R)	Ges. SR
1	19 / 7 / 4	82.6%	17 / 12 / 1	94.4%	36 / 19 / 5	87.8%
2	17 / 10 / 3	85.0%	21 / 5 / 4	84.0%	38 / 15 / 7	84.4%
3	23 / 1 / 6	79.3%	28 / 1 / 1	96.6%	51 / 2 / 7	87.9%
4	23 / 4 / 3	88.5%	27 / 2 / 1	96.4%	50 / 6 / 4	92.6%

Tabelle 3.3: Siegrate von AB+ID+MO+TT gegen einen Greedy-Spieler

Gegen den Zufallsspieler erreicht AB+ID+MO+TT über alle Suchtiefen hinweg eine Siegrate von $100\,\%$, unabhängig davon, ob als Spieler 1 oder Spieler 2 gespielt wird.

AB+ID+MO+TT erzielt gegenüber dem Greedy-Spieler unabhängig von der gewählten Suchtiefe hohe Siegraten. Bei Tiefe 1 und 2 liegen die Gesamtsiegraten bei 87,8 % bzw. 84,4 %. Bei Tiefe 3 beträgt die Siegrate 87,9 % und liegt damit nahezu auf dem Niveau von Tiefe 1. Bei Tiefe 4 wird mit 92,6 % die höchste Gesamtsiegrate erreicht. Spieler 2 erzielt dabei durchgehend höhere Siegraten als Spieler 1. Generell fällt auf, dass über alle Suchtiefen hinweg eine vergleichsweise hohe Anzahl an Unentschieden auftritt.

3.2 MCTS

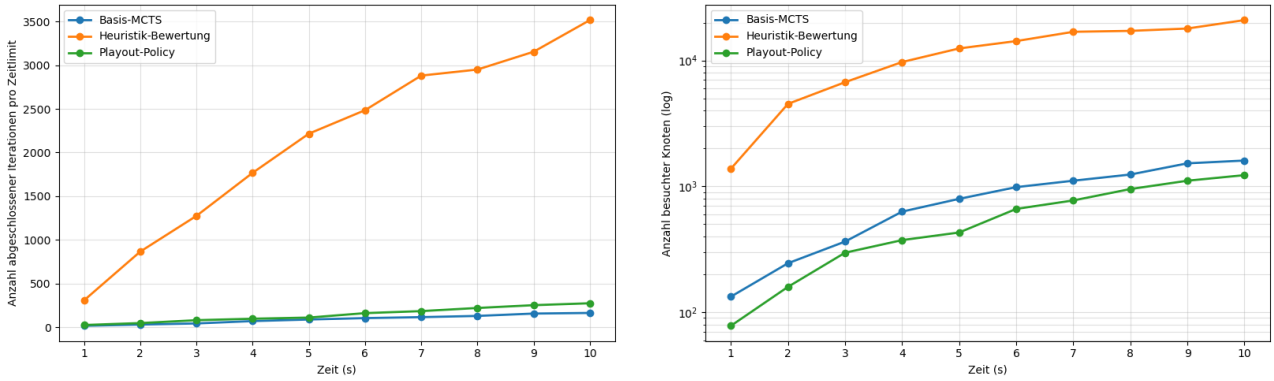
3.2.1 Test 1: Iterationen und besuchte Knoten bei festen Zeitlimits

Das Experiment untersucht die Rechengeschwindigkeit der drei MCTS-Modi, gemessen an der Anzahl verarbeiteter Iterationen und besuchter Knoten innerhalb eines festen Zeitbudgets. Um positionsbedingte Varianz auszuschließen, wurde eine feste Anfangsstellung verwendet.

Verglichen wurden die folgenden drei Varianten:

- **Basis-MCTS**: Monte Carlo Tree Search mit zufälligen Rollouts
- **Heuristik-Bewertung**: Monte Carlo Tree Search mit heuristikbasierter Zustandsbewertung
- **Playout-Policy**: Monte Carlo Tree Search mit strukturierter Playout-Policy

Für jedes Zeitlimit von 1 bis 10 Sekunden wurde der Algorithmus zehnmal ausgeführt und jeweils der Mittelwert gebildet, um stochastische Schwankungen auszugleichen. Erfasst wurden die Anzahl abgeschlossener Iterationen sowie die Anzahl besuchter Knoten.



(a) Anzahl abgeschlossener Iterationen pro Zeitlimit

(b) Anzahl besuchter Knoten pro Zeitlimit

Abbildung 3.4: MCTS-Varianten im Vergleich nach Zeitlimit

Die heuristikbasierte Bewertung führt mit Abstand die meisten Iterationen durch und besucht zugleich die meisten Knoten. Bei einem Zeitlimit von 10 Sekunden erreicht sie 3.516 Iterationen und 20.947 besuchte Knoten. Basis-MCTS und die Playout-Policy Variante liegen deutlich darunter. Basis-MCTS erzielt bei einem Zeitlimit von 10 Sekunden 164 Iterationen und 1.604 besuchte Knoten, während die Playout-Policy 275 Iterationen und 1.228 besuchte Knoten erreicht. Alle drei Modi zeigen mit wachsendem Zeitbudget einen annähernd linearen Anstieg. Dabei bleibt der Abstand zwischen der heuristikbasierten Bewertung und den übrigen Modi auch bei zunehmender Laufzeit weitgehend konstant. Auffällig ist, dass die Playout-Policy trotz mehr Iterationen als Basis-MCTS weniger Knoten besucht. Basis-MCTS hingegen besucht bei weniger Iterationen mehr Knoten.

3.2.2 Test 2: Vergleich der MCTS-Auswertungsstrategien

Bevor die leistungsstärkste MCTS-Variante gegen externe Referenzgegner getestet wurde, erfolgte zunächst ein direkter Vergleich der drei Varianten untereinander. Basis-MCTS, heuristikbasierte Bewertung und Playout-Policy traten dabei jeweils gegeneinander an. Spiele gegen sich selbst wurden jedoch nicht durchgeführt. Alle Varianten verwenden identische Parameter, einen Explorationskoeffizienten von $C = 1{,}41$ sowie ein Zeitlimit von 2 Sekunden pro Zug. Sie unterscheiden sich ausschließlich in der Art, wie der Simulations- bzw. Auswertungsschritt umgesetzt wird.

Für jede Variante wurden 60 Partien durchgeführt, je 30 als Spieler 1 und 30 als Spieler 2. Die Startpositionen wurden durch jeweils zehn zufällige Eröffnungszüge erzeugt, um eine größere Vielfalt an Spielsituationen zu gewährleisten. Ziel des Experiments ist es, die leistungsstärkere Variante für die nachfolgenden Tests gegen den Zufalls- und den Greedy-Spieler zu bestimmen.

Variante	Heuristik-Bewertung	Playout-Policy	Basis-MCTS
Heuristik-Bewertung	X	83.3%	93.3%
Playout-Policy	13.3%	X	56.7%
Basis-MCTS	3.3%	36.7%	X

Tabelle 3.4: Siegraten der MCTS-Varianten im direkten Vergleich

3.2.3 Test 3: Spielstärke der MCTS-Variante gegen Referenzgegner

Analog zu den Alpha-Beta-Experimenten wurde die heuristikbasierte MCTS-Variante zur Bewertung der praktischen Spielstärke gegen einen Zufallsspieler und einen Greedy-Spieler als Referenzgegner getestet.

Für beide Tests wurden 100, 300, 500 und 1000 Iterationen pro Zug verwendet. Für jeden Iterationswert wurden jeweils 30 Partien als Spieler 1 und 30 als Spieler 2 aus der Standardanfangsstellung gespielt.

Gegen den Greedy-Spieler wurden 30 zufällig generierte Startpositionen verwendet, die durch jeweils zehn zufällige Eröffnungszüge entstanden. In beiden Tests betrug das Zuglimit 200 Züge. Wie bereits bei Alpha-Beta gegen Referenzgegner in Abschnitt 3.1.4 gilt auch hier dieselbe Regel. Wird diese Grenze erreicht, ohne dass ein regulärer Sieg festgestellt werden konnte, endet die Partie unentschieden.

Iterationen	P1 (S/U/R)	P1 SR	P2 (S/U/R)	P2 SR	Ges. (S/U/R)	Ges. SR
100	30 / 0 / 0	100.0%	28 / 0 / 2	96.7%	58 / 0 / 2	96.7%
300	29 / 0 / 1	98.3%	30 / 0 / 0	100.0%	59 / 0 / 1	98.3%
500	30 / 0 / 0	100.0%	30 / 0 / 0	100.0%	60 / 0 / 0	100.0%
1000	30 / 0 / 0	100.0%	30 / 0 / 0	100.0%	60 / 0 / 0	100.0%

Tabelle 3.5: Siegrate von MCTS gegen einen Zufallsspieler

Iterationen	P1 (S/U/R)	P1 SR	P2 (S/U/R)	P2 SR	Ges. (S/U/R)	Ges. SR
100	21 / 0 / 9	70.0%	22 / 0 / 8	73.3%	43 / 0 / 17	71.7%
300	24 / 0 / 6	80.0%	26 / 0 / 4	86.7%	50 / 0 / 10	83.3%
500	27 / 0 / 3	90.0%	27 / 0 / 3	90.0%	54 / 0 / 6	90.0%
1000	28 / 1 / 1	96.6%	27 / 0 / 3	90.0%	55 / 1 / 4	93.2%

Tabelle 3.6: Siegrate von MCTS gegen einen Greedy-Spieler

Gegen den Zufallsspieler erzielt die MCTS-Variante bei 100 Iterationen $96{,}7\,\%$ und bei 300 Iterationen eine Gesamtsiegrate von $98{,}3\,\%$. Ab 500 Iterationen wird eine Siegrate von $100\,\%$ erreicht, die auch bei 1.000 Iterationen gehalten wird.

Die Ergebnisse zeigen, dass MCTS bei allen getesteten Iterationszahlen besser als der Greedy-Spieler abschneidet. Bei 100 Iterationen liegt die Siegrate bei 71,7 %. Mit steigender Iterationsanzahl nimmt sie zu. Bei 300 Iterationen erreicht sie 83,3 %, bei 500 Iterationen 90,0 % und bei 1000 Iterationen 93,2 %. Unentschieden treten insgesamt kaum auf, lediglich bei 1000 Iterationen wird eine unentschiedene Partie festgestellt.

3.2.4 Test 4: Bestimmung des Explorationsparameters

Zur Bestimmung des optimalen Explorationsparameters C wurde die heuristikbasierte MCTS-Variante mit sechs verschiedenen Werten ($C \in \{0{,}1,\ 0{,}5,\ 1{,}0,\ 1{,}41,\ 2{,}0,\ 3{,}0\}$) bei einer festen Iterationsanzahl von 500 Iterationen pro Zug gegen den Greedy-Spieler getestet. Für jeden C-Wert wurden 30

Startstellungen jeweils als erster und als zweiter Spieler gespielt, sodass sich insgesamt 60 Partien pro Parameterwert ergeben. Der Parameterwert mit der höchsten Gesamt-Siegrate wird anschließend im finalen Vergleich der MCTS-Variante gegen den Alpha-Beta-Algorithmus verwendet.

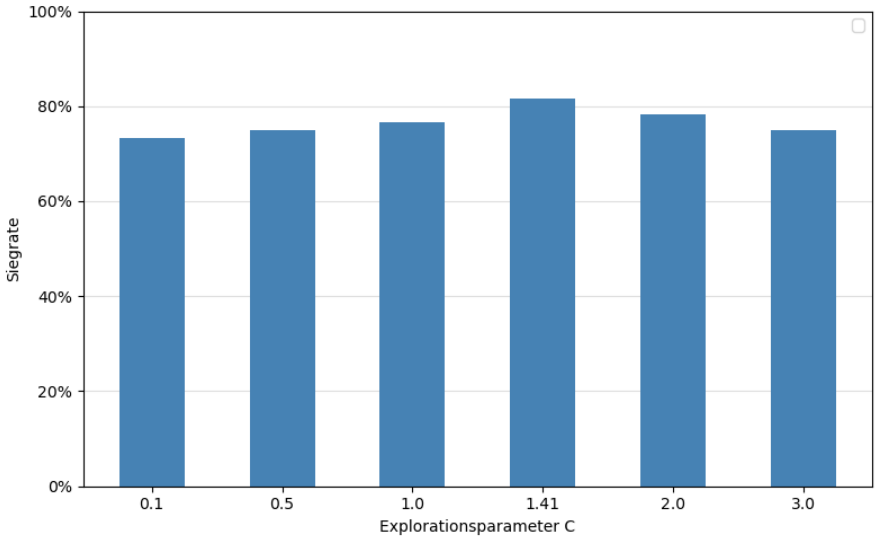


Abbildung 3.5: Siegrate von MCTS gegen den Greedy-Spieler in Abhängigkeit des Explorationsparameters C

Bei $C = 1{,}41$ wird mit $81{,}7\,\%$ die höchste Gesamtsiegrate erzielt. Mit steigendem C nimmt die Siegrate bei $C = 2{,}0$ ($78{,}3\,\%$) und $C = 3{,}0$ ($75{,}0\,\%$) wieder ab, während $C = 0{,}1$ mit $73{,}3\,\%$ den niedrigsten Wert aufweist. Für die nachfolgenden Experimente wird daher $C = 1{,}41 \approx \sqrt{2}$ verwendet.

3.3 Direkter Vergleich von Alpha-Beta und MCTS

Im abschließenden Experiment treten die beiden leistungsstärksten Varianten direkt gegeneinander an, nämlich AB+ID+MO+TT gegen die heuristikbasierte MCTS-Variante mit dem Explorationsparameter $C = 1{,}41$.

Um einen praxisnahen Vergleich zu ermöglichen, erhalten beide Algorithmen dasselbe Zeitbudget pro Zug. Damit wird untersucht, welche Spielstärke die Verfahren unter identischen Zeitbudgets erreichen. Dabei hängt das Ergebnis sowohl von der Leistungsfähigkeit des Algorithmus als auch von der Effizienz seiner Implementierung ab. Die Zeitlimits wurden auf 1 bis 10 Sekunden pro Zug gesetzt, um zu untersuchen, ob einer der Algorithmen stärker von zusätzlicher Rechenzeit profitiert.

Alle Partien beginnen von der Startaufstellung. Pro Zeitlimit werden 50 Partien mit AB+ID+MO+TT als Spieler 1 und 50 Partien mit MCTS als Spieler 1 gespielt, sodass sich 100 Partien pro Zeitlimit ergeben. Ebenso gilt in diesem Experiment ein Zuglimit von 200 Zügen. Bei Erreichen des Limits werden erneut dieselbe Regel sowie dieselbe Formel verwendet. Partien, die aufgrund des Zuglimits unentschieden enden, werden bei der Berechnung der Siegrate nicht berücksichtigt. Das Erreichen des Zuglimits ist keinem der beiden Spieler zuzurechnen, da es ausschließlich auf die festgelegte

maximale Zuganzahl zurückzuführen ist. Die Berechnung stellt sicher, dass durch das Erreichen des Zuglimits verursachte Unentschieden das Ergebnis nicht verzerren.

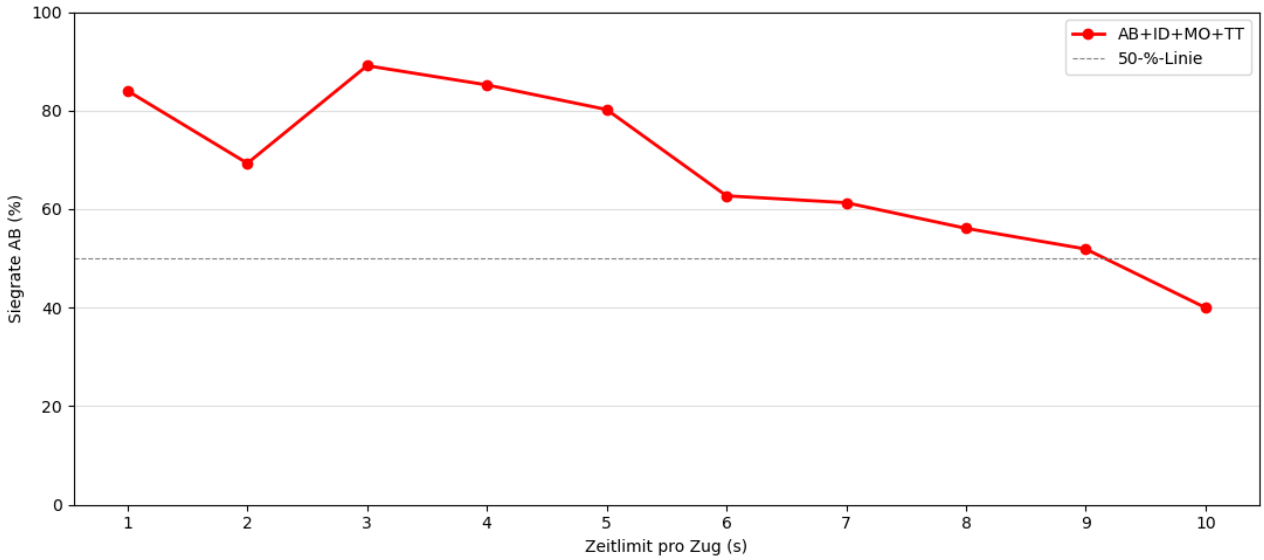


Abbildung 3.6: Direkter Vergleich der Siegraten nach Zeitlimit

Zeitlimit (s)	Siege AB	Siege MCTS	Unentschieden	SR
1	79	15	6	84.0%
2	61	27	12	69.3%
3	82	10	8	89.1%
4	75	13	12	85.2%
5	65	16	19	80.2%
6	52	31	17	62.7%
7	49	31	20	61.3%
8	46	36	18	56.1%
9	41	38	21	51.9%
10	30	45	25	40.0%

Tabelle 3.7: Ergebnisse von AB+ID+MO+TT im Vergleich zur heuristikbasierten MCTS-Variante in Abhängigkeit vom Zeitlimit

Die Siegraten zeigen einen klaren Zusammenhang zwischen Zeitlimit und Leistungsfähigkeit der beiden untersuchten Verfahren. AB+ID+MO+TT weist bei kurzen Zeitlimits die höheren Erfolgswerte auf und dominiert im Bereich von 1 bis 5 Sekunden. Die höchste Siegrate liegt bei 3 Sekunden und beträgt 89,1 %. MCTS mit Heuristik erzielt im selben Bereich niedrigere Werte und bleibt zunächst unterlegen.

Mit zunehmendem Zeitlimit verringert sich dieser Vorsprung kontinuierlich. Ab 6 Sekunden steigt die Siegrate von MCTS mit Heuristik an, während die Werte von AB+ID+MO+TT schrittweise abnehmen. Bei 9 Sekunden ist nahezu ein Gleichstand zwischen beiden Verfahren erreicht. Bei 10 Sekunden

fällt die Siegrate von AB+ID+MO+TT schließlich auf 40,0 % und liegt damit erstmals unter der von MCTS mit Heuristik. Gleichzeitig ist zu berücksichtigen, dass die Unentschieden-Rate von 6 % bei 1 Sekunde auf 25 % bei 10 Sekunden ansteigt, was die Interpretation der Ergebnisse bei höheren Zeitlimits einschränkt.

Die Ergebnisse legen nahe, dass AB+ID+MO+TT bei geringen Zeitbudgets effizient arbeitet, während MCTS stärker von längeren Rechenzeiten profitiert und seine Leistungsfähigkeit erst bei höheren Zeitlimits deutlicher sichtbar wird.

4 Diskussion

4.1 Alpha-Beta-Suche

Die Experimente zeigen, dass die Kombination AB+ID+MO+TT den größten Effizienzgewinn erzielt (vgl. Abb. 3.1 und Abb. 3.2). Dass der Vorteil mit zunehmender Suchtiefe größer wird, lässt sich durch die stärkere Wirkung von Transpositionstabelle und Zugsortierung in tieferen Suchläufen erklären. Je tiefer gesucht wird, desto größer wird der Suchraum. Dadurch können bereits berechnete Stellungen häufiger erneut erreicht werden, während eine gute Zugsortierung zusätzliche Abschneidungen begünstigt.

Auf den ersten Blick scheinen die Ergebnisse aus Abb. 3.2 den Befunden aus Abb. 3.1 im Vergleich von AB und AB+ID+MO teilweise zu widersprechen. Es wirkt überraschend, dass AB+ID+MO unter festen Zeitlimits ähnliche Suchtiefen erreicht wie AB (vgl. Abb. 3.2), obwohl es in Abb. 3.1 teils eine moderate aber dennoch Knotenreduktion gegenüber AB aufweist. Ein unmittelbarer Widerspruch liegt jedoch nicht vor, da beiden Abbildungen unterschiedliche Messdesigns und unterschiedliche Stichproben zugrunde liegen.

In Abb. 3.1 wird die Laufzeit bei fester Suchtiefe auf einer kleinen Menge fester Teststellungen gemessen. In einem solchen Setting kann der beobachtete Vorteil von AB+ID+MO gegenüber AB von den konkreten Eigenschaften dieser Positionen abhängen. Weisen die gewählten Stellungen eine Struktur auf, in der die Zugsortierung früh günstige Kandidaten priorisiert, kann Alpha-Beta stärker schneiden und dadurch eine moderate Reduktion der effektiven Sucharbeit zeigen.

Abb. 3.2 erfasst dagegen eine andere Zielgröße, nämlich die unter einem festen Zeitbudget erreichbare maximale Suchtiefe. Zudem basiert diese Auswertung auf einer größeren Menge zufällig generierter, durch feste Seeds reproduzierbarer Stellungen aus mehreren Spielphasen. Dadurch ist die Stichprobe vielfältiger als die kleine Menge fester Teststellungen aus Abb. 3.1. In einem solchen breiteren Mittel können sich stellungsspezifische Vorteile der Zugsortierung teilweise abschwächen. Dadurch ist es nachvollziehbar, dass AB und AB+ID+MO bei der erreichten Tiefe näher zusammenliegen, obwohl in einem engeren festen Positionssatz moderate Unterschiede sichtbar werden.

Hinzu kommt, dass unterschiedliche Positionen den Verzweigungsfaktor und die Struktur des Suchraums beeinflussen können, sodass sich die Ergebnisse beider Experimente nur eingeschränkt direkt miteinander vergleichen lassen. Ein weiterer möglicher Grund liegt darin, dass die iterative Tiefensuche allein zunächst zusätzlichen Overhead verursacht, da bereits betrachtete Suchtiefen mehrfach untersucht werden. Reinefeld und Marsland (1994) [8] zeigen, dass dieser Overhead mit wachsendem Verzweigungsfaktor abnimmt, sich jedoch ohne ergänzende Verbesserungen über alle Iterationen hinweg summiert. Ohne die ergänzende Wirkung der Transpositionstabelle kann dieser Overhead nicht kompensiert werden. Erst in Kombination, wie bei AB+ID+MO+TT, überwiegen die Vorteile der iterativen Tiefensuche den zusätzlichen Aufwand. Insgesamt spricht dies eher für eine stellungs- und

metrikabhängige Wirkung von AB+ID+MO als für einen tatsächlichen Widerspruch zwischen beiden Abbildungen.

Der spielphasenspezifische Knotenzahlvergleich in Abb. 3.3 zeigt, dass die Wirkung der untersuchten Erweiterungen von der jeweiligen Stellung abhängt. Die deutlich stärkere Knotenreduktion in der Endsituation dürfte vor allem mit der Struktur des Endspiels zusammenhängen (vgl. Abb. 3.3 (c)). Da in dieser Phase meist weniger Züge verfügbar sind und der Suchraum stärker eingeschränkt ist, können bereits berechnete Stellungen häufiger erneut auftreten, wodurch die Transpositionstabelle wirksam wird. Zudem begünstigt eine klarere Endspielstruktur die Zugsortierung, was frühere Alpha-Beta-Abschneidungen ermöglicht. Im Mittelspiel (vgl. Abb. 3.3 (b)) zeigt sich hingegen ein differenzierteres Bild. Während Varianten mit Transpositionstabelle bereits weniger Knoten als die Basisvariante besuchen, liegt AB+ID+MO weiterhin über AB. Dies lässt sich damit erklären, dass die durch iterative Tiefensuche verbesserte Zugsortierung allein noch nicht ausreicht, um den zusätzlichen Rechenaufwand zu kompensieren. Ohne Transpositionstabelle werden identische oder ähnliche Stellungen weiterhin mehrfach berechnet, sodass der Nutzen der verbesserten Zugreihenfolge begrenzt bleibt. Erst durch die Kombination mit der Transpositionstabelle kann dieser Effekt vollständig zum Tragen kommen. In der Endsituation greifen diese Mechanismen schließlich zusammen. Hier reicht die verbesserte Zugsortierung aus, um zusammen mit der reduzierten Komplexität des Suchraums auch ohne Transpositionstabelle eine Verringerung der Knotenzahl gegenüber AB zu erzielen. In Kombination mit der Transpositionstabelle wird dieser Effekt weiter verstärkt, sodass AB+ID+MO+TT die geringste Knotenzahl aufweist. Zu beachten ist zudem, dass die Ergebnisse von der Wahl der jeweiligen Teststellung abhängen. Da pro Spielphase nur eine einzelne Position betrachtet wird, sind die Befunde nicht zwingend repräsentativ für alle möglichen Stellungen dieser Phase.

Die Siegrate von 100 % gegen den Zufallsspieler ist erwartungsgemäß und dient in erster Linie als Grundvalidierung (vgl. Tab. 3.2). Interessanter ist jedoch der Vergleich mit dem Greedy-Spieler. Die dort durchgehend über 80 % liegenden Siegraten sprechen für eine insgesamt robuste Spielstärke (vgl. Tab. 3.3). Auffällig ist jedoch, dass die Ergebnisse zwischen den Suchtiefen 1 bis 3 kein monotones Muster zeigen. Während die Gesamtsiegrate bei Suchtiefe 2 mit 84,4 % am niedrigsten ausfällt, liegt sie bei Suchtiefe 3 mit 87,9 % wieder auf dem Niveau von Suchtiefe 1. Die Unterschiede lassen sich daher nicht allein durch die zunehmende Suchtiefe erklären. Hinzu kommt, dass sich die Siegraten je nach Spielerrolle unterschiedlich entwickeln. Während als Spieler 1 die niedrigste Siegrate bei Suchtiefe 3 auftritt, zeigt sich als Spieler 2 der schwächste Wert bei Suchtiefe 2. Dies spricht dafür, dass neben der Suchtiefe auch rollen- und stellungsspezifische Effekte eine Rolle spielen. Zudem ist zu berücksichtigen, dass ein Teil der Partien als Unentschieden endet, da das Zuglimit von 200 Zügen erreicht wurde. Da Unentschieden bei der Berechnung der Siegrate nicht berücksichtigt werden, führt eine höhere Anzahl an Timeouts dazu, dass die Siegrate auf einer kleineren Grundgesamtheit entschiedener Partien berechnet wird, was ihre Aussagekraft einschränkt.

4.2 Monte Carlo Tree Search

Der deutliche Unterschied in der Anzahl der Iterationen zwischen den drei Modi lässt sich durch den unterschiedlichen Rechenaufwand der jeweiligen Simulationsstrategie erklären (vgl. Abb. 3.4). Basis-MCTS folgt dem klassischen Ablauf aus Selection, Expansion, Simulation und Backpropagation und führt zufällige Simulationen bis zum Spielende durch. Die Basisvariante verwendet bewusst

keine Lazy Expansion, um eine direkte Vergleichsbasis zum klassischen MCTS-Verfahren zu erhalten. Heuristik-Bewertung und Playout-Policy nutzen hingegen Lazy Expansion, bei der die legalen Aktionen eines Knotens nicht bereits bei der Knotenerstellung vollständig berechnet, sondern erst beim ersten Expansionsaufruf initialisiert werden. Bei weiteren Besuchen wird dann jeweils nur ein weiterer Zug expandiert. Dadurch reduziert sich insbesondere in Quoridor der Expansionsaufwand pro Iteration, da die Berechnung legaler Wandzüge aufgrund der erforderlichen Astar-Prüfungen besonders kostenintensiv ist. Die heuristikbasierte Bewertung verzichtet zudem auf Simulationen bis zum Spielende und bewertet stattdessen die aktuelle Stellung direkt. Dadurch wird pro Iteration deutlich weniger Rechenzeit benötigt, sodass innerhalb desselben Zeitbudgets mehr Iterationen durchgeführt werden können.

Die Playout-Policy verfolgt demgegenüber einen anderen Ansatz. Im Unterschied zur heuristikbasierten Bewertung führt sie weiterhin vollständige Simulationen bis zum Spielende durch. Dabei werden entlang der simulierten Partie zahlreiche aufeinanderfolgende Stellungen durchlaufen und bewertet, sodass die resultierenden Bewertungen auf tatsächlich ausgespielten Partieverläufen basieren und stabilere Schätzungen der Erfolgswahrscheinlichkeit ermöglichen. Dem steht jedoch ein erhöhter Rechenaufwand gegenüber, da die strukturierte, BFS-basierte Zugauswahl innerhalb der Simulation wiederholt zusätzliche Berechnungen erfordert.

Auffällig ist, dass die Playout-Policy trotz mehr Rollouts als Basis-MCTS weniger Knoten besucht. Ein möglicher Grund dafür ist, dass die Playout-Policy durch ihre BFS-basierte Simulation zielgerichteter vorgeht und dadurch im Mittel kürzere Spielverläufe erzeugt. Das Spielende wird somit auf direkterem Weg erreicht. Basis-MCTS simuliert hingegen zufällig, was tendenziell längere Playouts und damit mehr besuchte Knoten pro Rollout zur Folge haben kann, obwohl insgesamt weniger Rollouts abgeschlossen werden.

Der direkte Vergleich der MCTS-Auswertungsstrategien zeigt deutliche Unterschiede zwischen den drei untersuchten MCTS-Varianten (vgl. Tab. 3.4). Besonders auffällig ist die Überlegenheit der heuristikbasierten Bewertung. Sie erzielt gegenüber der Playout-Policy eine Siegrate von $83{,}3\,\%$ und gegenüber Basis-MCTS sogar $93{,}3\,\%$. Auch die Playout-Policy ist dem Basis-MCTS mit einer Siegrate von $56{,}7\,\%$ überlegen. Ein möglicher Grund für dieses Ergebnis liegt im unterschiedlichen Rechenaufwand der Simulationsstrategien. Basis-MCTS verwendet zufällige Rollouts, die zwar einfach umzusetzen sind, jedoch eine hohe Varianz aufweisen und vergleichsweise wenig zielgerichtete Informationen liefern. Die Playout-Policy reduziert diese Zufälligkeit durch eine strukturierte, BFS-basierte Simulation, was zu potenziell aussagekräftigeren Playouts führen kann, jedoch mit zusätzlichem Rechenaufwand verbunden ist. Gleichzeitig ist zu berücksichtigen, dass aufgrund der verwendeten Lazy Expansion keine gleichermaßen fundierte Bewertung aller möglichen Fortsetzungen erfolgt, da nicht alle Folgezüge eines Knotens unmittelbar expandiert und berücksichtigt werden. Die heuristikbasierte Bewertung verzichtet dagegen vollständig auf Simulationen bis zum Spielende und ersetzt den Simulationsschritt durch eine direkte Bewertung der aktuellen Stellung. Gerade dieser Verzicht auf Rollouts erweist sich unter den gewählten Versuchsbedingungen als entscheidender Vorteil, da dadurch innerhalb desselben Zeitbudgets deutlich mehr Iterationen durchgeführt werden können. Dieser Effizienzvorteil schlägt sich auch in der Spielstärke nieder, da die heuristikbasierte Variante die höchsten Siegraten erzielt. Zugleich ist zu beachten, dass ein solcher Verzicht auf Rollouts vom klassischen MCTS-Ansatz abweicht, in dem Simulationen bis zu terminalen Zuständen üblicherweise ein zentrales Element darstellen [2]. Der Verzicht wurde hier bewusst implementiert, um den Rechenaufwand

pro Iteration zu reduzieren. Dies ist für Quoridor relevant, da vollständige Simulationen aufgrund des hohen Verzweigungsfaktors und der kostenintensiven Zugberechnung mit erheblichem Zeitaufwand verbunden sind.

Ähnlich wie in den Alpha-Beta-Experimenten ist die Spielstärke gegen den Zufallsspieler erwartungsgemäß hoch (vgl. Tab. 3.5). Wesentlich interessanter ist auch hier der Vergleich mit dem Greedy-Spieler. Die Siegrate steigt von $71{,}7\,\%$ bei 100 Simulationen auf $93{,}2\,\%$ bei 1.000 Simulationen, was darauf hindeutet, dass eine größere Anzahl an Simulationen die Zugbewertung verlässlicher macht (vgl. Tab. 3.6). Obwohl Alpha-Beta und MCTS in den Experimenten dieselbe heuristische Bewertungsfunktion verwenden, ist eine unterschiedliche Anzahl an Zuglimit-Timeouts erwartbar, da beide Verfahren die Heuristik in unterschiedlichen Entscheidungskontexten nutzen. Bei Alpha-Beta dient sie vor allem der Bewertung von Blattknoten innerhalb eines einzelnen, deterministischen Suchlaufs mit begrenztem Horizont. Sind mehrere Alternativen nahezu gleich bewertet, kann Alpha-Beta Züge wählen, die keine klare Fortschrittsrichtung erzeugen. Dadurch können Hin-und-her-Bewegungen oder allgemein zyklische Verläufe entstehen, die das Erreichen des Zuglimits begünstigen. MCTS verwendet dieselbe Heuristik dagegen eingebettet in viele iterative Suchdurchläufe. Einzelbewertungen werden dabei nicht isoliert verwendet, sondern über zahlreiche Iterationen statistisch aggregiert. Dadurch können lokale Schwankungen teilweise ausgeglichen werden. Zusätzlich basiert die Zugauswahl bei MCTS auf aggregierten Suchstatistiken, während Alpha-Beta eine deterministische Entscheidung aus einem einzelnen Suchbaum ableitet.

Der optimale Explorationsparameter $C = 1{,}41 \approx \sqrt{2}$ deckt sich mit dem in der Literatur häufig genannten Standardwert (vgl. Abb. 3.5) [4] . Die sinkende Siegrate bei höheren C-Werten legt nahe, dass in Quoridor eine zu starke Exploration nachteilig ist. Da das Spiel durch einen hohen Verzweigungsfaktor gekennzeichnet ist, kann eine stärkere Ausrichtung auf aussichtsreiche Züge effektiver sein als eine noch breitere Exploration.

4.3 Direkter Vergleich von Alpha-Beta und MCTS

Unter kurzen Zeitlimits erweist sich AB+ID+MO+TT als das stärkere Verfahren (vgl. Abb. 3.6) und dominiert den Bereich von 1 bis 5 Sekunden. Die heuristikbasierte MCTS-Variante erzielt in diesem Intervall geringere Siegraten und bleibt zunächst unterlegen. Mit wachsendem Zeitbudget verringert sich der Vorsprung von AB+ID+MO+TT jedoch zunehmend (vgl. Tab. 3.7). MCTS profitiert zugleich stärker von zusätzlicher Rechenzeit, da mit wachsender Iterationsanzahl die statistische Einschätzung der Zugqualität stabiler wird. Ab 6 Sekunden verbessert sich die Erfolgsrate von MCTS, während die Werte von AB+ID+MO+TT schrittweise zurückgehen.

Alpha-Beta zeigt bei den meisten Zeitlimits die besseren Ergebnisse als MCTS. Auffällig ist jedoch, dass mit zunehmendem Zeitlimit die Zahl der Unentschieden beziehungsweise Timeouts ansteigt, während die Siegrate von Alpha-Beta sinkt. Dies deutet darauf hin, dass längere Rechenzeiten nicht ausschließlich zu klareren Gewinnentscheidungen führen, sondern vermehrt auch zu lang andauernden Partieverläufen beitragen. Intuitiv wäre zu erwarten, dass mit größerem Zeitbudget die höhere Suchtiefe von Alpha-Beta zu eindeutigeren Entscheidungen und damit zu weniger Zuglimit-Timeouts führt. Die Ergebnisse zeigen jedoch, dass dieser Zusammenhang nicht zwingend gilt.

Eine mögliche Erklärung hierfür liegt in der deterministischen Entscheidungsstruktur von Alpha-

Beta. In Stellungen, die durch die Heuristik ähnlich bewertet werden, kann es dazu kommen, dass zwischen mehreren nahezu gleichwertigen Zügen gewechselt wird, ohne dass eine eindeutige Fortschrittsrichtung entsteht. Dadurch können ähnliche oder wiederkehrende Stellungen mehrfach erreicht werden, was die Partie verlängert und das Erreichen des Zuglimits begünstigt. Solche Verläufe können auch damit zusammenhängen, dass Alpha-Beta versucht, dem Gegner den Weg zum Ziel möglichst lange zu erschweren, ohne dabei selbst einen klar gewinnbringenden Pfad zu finden. Auch qualitative Beobachtungen einzelner Spielverläufe sprechen tendenziell für eine solche Interpretation.

Die mit zunehmendem Zeitlimit steigende Zahl an Unentschieden beziehungsweise Timeouts legt daher nahe, dass solche schwer aufzulösenden Entscheidungssituationen eine wichtige Rolle spielen. Zwar ist nicht auszuschließen, dass auch allgemein lange Partieverläufe zu einem Teil dieser Timeouts beitragen, die Ergebnisse sprechen jedoch dafür, dass wiederholte Zustandsmuster und eng beieinanderliegende heuristische Bewertungen hierbei relevant sein könnten. Insgesamt bleibt festzuhalten, dass AB+ID+MO+TT im direkten Vergleich bei den meisten Zeitlimits besser abschneidet als MCTS. Dieses Ergebnis sollte jedoch mit einer gewissen Vorsicht interpretiert werden, da die konkrete Leistungsfähigkeit nicht nur vom Verfahren selbst, sondern auch von der Qualität seiner Implementierung und der verwendeten Heuristik abhängt. Die vorliegenden Befunde sprechen daher für einen Vorteil von Alpha-Beta unter den gewählten Versuchsbedingungen, zeigen aber zugleich, dass bei lang andauernden, schwer aufzulösenden Partieverläufen noch Optimierungspotenzial besteht.

4.4 Limitationen der Arbeit

Die vorliegende Arbeit unterliegt einigen Einschränkungen, die bei der Interpretation der Ergebnisse zu berücksichtigen sind.

Eine wesentliche Einschränkung dieser Arbeit liegt in der Effizienz der Implementierung, da bei beiden Algorithmen derselbe rechenintensive Verarbeitungsschritt anfällt. Die Astar-basierte Wandvalidierung fällt bei Alpha-Beta an jedem Suchknoten und bei MCTS bei jeder Expansion an. Bei einem Verzweigungsfaktor von etwa 60 möglichen Zügen pro Stellung addiert sich dieser Aufwand. Hinzu kommt, dass die Heuristik an Blattknoten mehrere BFS-Pfaddistanzberechnungen für beide Spieler benötigt (teils mit Caching, teils mit zusätzlicher Suche nahe am Ziel) und der Spielzustand in vielen Schritten als Kopie eines 289-Element-Boards angelegt wird, etwa bei der Erzeugung von Kindknoten bzw. in Expansion und Iterationsschritte.

Dies erklärt, warum MCTS selbst bei 10 Sekunden nur rund 3.500 Iterationen erreicht. Bei einer effizienteren Implementierung würde die Zahl höher ausfallen und die Spielstärke von MCTS steigern. Die Wahl von Python als Implementierungssprache verstärkt diesen Effekt zusätzlich, da kompilierte Sprachen wie C++ erfahrungsgemäß einen großen Geschwindigkeitsvorteil bieten. Eine naheliegende Verbesserung wäre der Einsatz einer bitboard-basierten Zustandsrepräsentation [1], bei der Spielfelder, Spielerpositionen und Wandbelegungen als Bitmuster in mehreren Maschinenwörtern kodiert werden. Dadurch ließen sich Zustandskopien auf wenige Wortzuweisungen reduzieren. Zudem könnten Erreichbarkeits- bzw. Distanzberechnungen teilweise über bitweise Flood-Fill-Operationen beschleunigt werden, was den Aufwand gegenüber der aktuellen suchebasierten Validierung verringern kann.

[1] https://www.chessprogramming.org/Bitboards Zugriff: 29.03.2026

Eine weitere Einschränkung betrifft die verwendete Bewertungsfunktion. Diese basiert auf lediglich vier vergleichsweise einfachen Merkmalen mit fest vorgegebenen Gewichten. Die Entwicklung geeigneter Features ist jedoch keine triviale Aufgabe, da ein gutes Merkmal nicht nur die aktuelle Stellung bewerten, sondern auch Aussagen über den weiteren Spielverlauf ermöglichen sollte, ohne zugleich zu rechenintensiv zu sein. Auch die Kalibrierung der Gewichte stellt ein grundlegendes Problem dar. Die verwendete Heuristik ist vergleichsweise einfach und erfasst möglicherweise nicht alle strategisch relevanten Spielsituationen mit ausreichender Genauigkeit. Eine systematische Gewichtsoptimierung, etwa mithilfe eines genetischen Algorithmus wie bei Glendenning (2005) [3], könnte die Qualität der verwendeten Bewertungsfunktion insgesamt verbessern. Da diese in der vorliegenden Implementierung sowohl in Alpha-Beta als auch in Teilen von MCTS eingesetzt wird, könnte eine solche Optimierung die Spielstärke beider Verfahren beeinflussen und damit auch das Ergebnis des direkten Vergleichs verändern.

Zudem wurde der isolierte Effekt einzelner Erweiterungen nicht vollständig untersucht. So wurde beispielsweise nicht systematisch analysiert, welchen Beitrag die Zugsortierung innerhalb von AB+ID unabhängig von anderen Optimierungen leistet. Eine feinere experimentelle Zerlegung der einzelnen Komponenten hätte hier zu einer noch präziseren Bewertung ihres jeweiligen Nutzens beitragen können.

Auch die Aussagekraft der verwendeten Referenzgegner ist begrenzt. Ein Vergleich mit menschlichen Spielern wurde nicht durchgeführt, sodass keine Aussage über die absolute Spielstärke der implementierten Agenten möglich ist. Da der Greedy-Spieler lediglich auf einfachen Entscheidungsregeln basiert, erlaubt die erzielte Siegrate gegen diesen Referenzgegner nur begrenzte Rückschlüsse auf die tatsächliche Spielstärke. Ein aussagekräftigerer Vergleich würde stärkere Referenzagenten oder menschliche Testspieler erfordern.

Ein gelegentlich in der grafischen Oberfläche beobachtetes Hin-und-her-Bewegen der Spielfiguren stellt ein weiteres Indiz für die begrenzte Trennschärfe der verwendeten Heuristik dar. Trotz der Berücksichtigung von Pfadlängen zum Ziel werden mehrere legale Züge häufig nahezu gleich bewertet. Bei MCTS kann die stochastische Schätzung unter begrenztem Iterationsbudget dazu führen, dass zwischen annähernd gleichwertigen Alternativen gewechselt wird, wobei dieser Effekt durch die statistische Aggregation tendenziell abgeschwächt wird. Bei Alpha-Beta entsteht ein ähnlicher Effekt durch geringe Bewertungsabstände in der Blattevaluation. In beiden Fällen kann dies eine stabile und durchgehend zielgerichtete Annäherung an das Ziel erschweren.

Für zukünftige Arbeiten bieten sich mehrere Ansatzpunkte an. Ein zentraler Aspekt betrifft die Weiterentwicklung der Heuristikfunktion. Diese könnte sowohl durch zusätzliche Merkmale als auch durch eine präzisere Gewichtung der bereits verwendeten Merkmale verbessert werden, um insbesondere strategisch komplexe Mittel- und Endspielsituationen besser zu erfassen. Zur Erweiterung der Heuristik könnte zudem Mustererkennung eingesetzt werden, bei der Muster aus Wand- und Figurenpositionen abgeleitet und mithilfe von Hebbschem Lernen trainiert werden [6]. Auch evolutionäre Algorithmen zur automatisierten Gewichtsoptimierung erscheinen vielversprechend, wie bereits Glendenning (2005) [3] gezeigt hat. Aus technischer Sicht wäre insbesondere eine Reimplementierung mit Bitboards sowie in einer kompilierten Sprache naheliegend, um die Recheneffizienz beider Algorithmen deutlich zu steigern. Langfristig könnte auch der Einsatz moderner Deep-Learning-Verfahren ein interessanter Ansatz sein, um sowohl die Stellungsbewertung als auch die Zugauswahl grundlegend zu verbessern. Ähnliche Ansätze wurden bereits in anderen Spielen eingesetzt, etwa im Rahmen von

AlphaGo, wo neuronale Netze zur Bewertung von Positionen und zur Steuerung der Suche verwendet werden [12].

4.5 Beitrag der Arbeit

Diese Arbeit leistet mehrere Beiträge zur algorithmischen Untersuchung von Quoridor. Es wurden mehrere Alpha-Beta-Varianten systematisch miteinander verglichen und der isolierte Einfluss einzelner Optimierungen wie Transpositionstabelle, iterativer Tiefensuche und Zugsortierung quantitativ belegt. Ergänzend wurden drei verschiedene MCTS-Simulationsstrategien evaluiert, wobei die heuristikbasierte Bewertung die stärksten Ergebnisse erzielte. Der abschließende direkte Vergleich beider Verfahren unter identischen experimentellen Bedingungen zeigt, dass AB+ID+MO+TT insgesamt die besseren Ergebnisse erzielt, während MCTS mit zunehmendem Rechenbudget an Leistungsfähigkeit gewinnt und bei höheren Zeitlimits langsam überholt.

5 Fazit

Diese Arbeit untersuchte die Frage, welche der beiden Suchmethoden Alpha-Beta und Monte Carlo Tree Search bei vergleichbarem Rechenbudget die höhere Spielstärke in Quoridor erzielt. Dazu wurden zunächst mehrere Varianten beider Algorithmen untereinander verglichen, bevor die jeweils leistungsstärkste Variante im direkten Duell gegenübergestellt wurde. Die Ergebnisse zeigen, dass Alpha-Beta unter den gewählten Bedingungen erwartungsgemäß starke Leistungen erzielt, während MCTS insbesondere durch die heuristikbasierte Variante überraschend konkurrenzfähig ist, obwohl keine klassischen Rollouts verwendet werden.

Die Ergebnisse zeigen, dass AB+ID+MO+TT unter den gewählten Versuchsbedingungen insgesamt die stärkeren Resultate erzielt. Insbesondere bei kurzen Zeitlimits erweist sich Alpha-Beta als überlegen, da bereits mit begrenzter Rechenzeit mehrere Züge im Voraus berücksichtigt werden können. Die heuristikbasierte MCTS-Variante profitiert hingegen stärker von zusätzlicher Rechenzeit, da innerhalb desselben Zeitbudgets deutlich mehr Iterationen durchgeführt werden können und sich die statistische Zugbewertung dadurch zunehmend verbessert. Mit wachsendem Zeitbudget verringert sich daher der Vorsprung von Alpha-Beta, und bei hohen Zeitlimits kann MCTS aufschließen beziehungsweise in einzelnen Fällen bessere Ergebnisse erzielen.

Insgesamt sprechen die Ergebnisse für einen Vorteil von Alpha-Beta unter den hier gewählten Versuchsbedingungen. Zugleich verdeutlicht die Arbeit, dass MCTS mit wachsendem Rechenbudget deutlich an Leistungsfähigkeit gewinnt. Die Eignung von Alpha-Beta und MCTS lässt sich daher nur im jeweiligen Anwendungskontext beurteilen, da ihre Leistungsfähigkeit wesentlich von den verfügbaren Rechenressourcen sowie der konkreten technischen Umsetzung abhängt.

Verwendete Hilfsmittel

Für die vorliegende Arbeit wurden verschiedene digitale Hilfsmittel unterstützend eingesetzt. Das generische KI-Tool ChatGPT von OpenAI wurde zur Grammatik- und Rechtschreibprüfung verwendet. Die Unterstützung bezog sich insbesondere auf sprachliche Überarbeitungen, darunter Kürzungen, Umformulierungen sowie die Formulierung von Titeln und Bildunterschriften. Darüber hinaus wurden Hinweise zur Gliederung, zur konsistenten Gestaltung von Tabellen und Abbildungen sowie programmiertechnische Hilfestellungen, insbesondere zu Python, Refactoring und LaTeX, genutzt. Einzelne durch ChatGPT vorgeschlagene Codefragmente wurden eigenständig geprüft, angepasst, getestet und verifiziert.

Zusätzlich wurde DeepL für sprachliche Überarbeitungen und Formulierungsvorschläge genutzt. Scribbr wurde zur Unterstützung bei Fragen der wissenschaftlichen Formulierung, Zitierweise und sprachlichen Überarbeitung herangezogen. Alle mit diesen Hilfsmitteln erarbeiteten Inhalte wurden eigenständig geprüft, überarbeitet und verantwortet.

Tabellenverzeichnis

Literaturverzeichnis

[1] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik. Information in Transposition Tables. In *Advances in Computer Chess 8*, volume 8, pages 199–212. Universiteit van Maastricht, 1997.

[2] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon 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.

[3] Lisa Glendenning. Mastering Quoridor. B.Sc. thesis, The University of New Mexico, Albuquerque, New Mexico, 2005.

[4] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293, Berlin, Germany, 2006. Springer.

[5] Richard E. Korf. Depth-First iterative-deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[6] Mark Hudson Beale Martin T. Hagan, Howard B. Demuth and Orlando De Jesús. *Neural Network Design*. Martin Hagan, Stillwater, Oklahoma, 2 edition, 2014.

[7] P.J.C. Mertens. A Quoridor-playing Agent. B.Sc. thesis, Maastricht University, 2006.

[8] Alexander Reinefeld and T. A. Marsland. Enhanced Iterative-Deepening Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[9] Victor Massagué Respall. Quoridor agent using Monte Carlo tree search. B.Sc. thesis, Innopolis University, 2018.

[10] Victor Massagué Respall, Joseph Alexander Brown, and Hamna Aslam. Monte Carlo tree search for Quoridor. In *19th International Conference on Intelligent Games and Simulation (GAME-ON 2018)*. EUROSIS, 2018.

[11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education Limited, Harlow, United Kingdom, 4 edition, 2022.

[12] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[13] Albert L. Zobrist. A New Hashing Method With Application for Game Playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1970.