Systematischer Vergleich von Alpha-Beta und Monte-Carlo Tree Search für das Spiel Abalone

Bachelorarbeit

Malte Hauff
402073

28.03.2024

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

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

Kurzfassung

Sowohl die Alpha-Beta-Suche als auch die Monte Carlo Tree Search sind bei der Entscheidungsfindung für den besten Spielzug etablierte Algorithmen. Je nach Spiel agieren die beiden Algorithmen allerdings verschieden erfolgreich. In dieser Arbeit werden sie in Bezug auf das kombinatorische Zwei-Spieler-Spiel Abalone untersucht.
Zunächst werden die Algorithmen und ihre Erweiterungen theoretisch eingeführt. Es werden sowohl Techniken aus bestehenden Forschungsergebnissen umgesetzt als auch neue analysiert. Anschließend werden diese durch Spielsimulationen systematisch verglichen und somit die erfolgreichsten Algorithmusvarianten ermittelt. Schließlich werden diese direkt in Simulationen gegeneinander gemessen. Für beide Algorithmen sind jeweils verbesserte Varianten herausgearbeitet worden. Im direkten Vergleich stellt sich die Alpha-Beta-Suche gegenüber dem Monte Carlo Tree Search-Algorithmus als deutlich überlegen heraus.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In den vergangen Jahren waren große Entwicklungen im Bereich der künstlichen Intelligenz zu beobachten. Auch auf dem Gebiet der Spieltheorie, die sich unter anderem mit der Entscheidungsfindung bezüglich Spielzügen beschäftigt, spielt die künstliche Intelligenz eine wichtige Rolle.[17] Da viele dieser Entscheidungsprobleme nur in einer exponentiellen Laufzeit zu lösen sind, bedarf es hier der Entwicklung von Algorithmen, die den Suchraum möglichst effizient und zielgerichtet durchsuchen.

Lange Zeit galt das heuristische Suchverfahren der Alpha-Beta-Suche als die erfolgreichste Methode zur Entwicklung künstlicher Intelligenzen für ein Spiel. [11]

Seitdem 2006 unter anderem von Rémi Coulom in seiner Arbeit „Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search“ [7] der Monte Carlo Tree Search-Algorithmus als neue Methode vorgestellt wurde, konnten viele künstliche Intelligenzen (KIs) entwickelt werden, die andere auf dem herkömmlichen Alpha-Beta-Ansatz basierende KIs schlagen. Als Beispiel wäre die für das Spiel *Go* von Google DeepMind entwickelte KI *AlphaGo* zu nennen.[22] Dagegen gilt unter anderem für das Strategiespiel Schach weiterhin die Alpha-Beta-Suche als die stärkste Methode. Welche der beiden Entscheidungsmethoden bei welchen Spielen erfolgreicher ist, bleibt weiter Gegenstand der Forschung.[6]

In dieser Arbeit wird das kombinatorische Zwei-Spieler [1]-Spiel Abalone untersucht. Bereits in den Arbeiten von Pascal Chorus [16], Michiel Verloop [19] und der Autorengruppe um Athanasios Papadopoulos [15] wurde für das Spiel eine Alpha-Beta-Suche umgesetzt und weiterentwickelt. Pascal Chorus vergleicht dabei diesen Ansatz bereits mit einer Monte-Carlo-Suche, die aber -anders als der Monte Carlo Tree Search-Algorithmus- ohne Hilfe eines Suchbaums arbeitet.

Das Ziel dieser Arbeit ist es, die bestehenden Ergebnisse aus der Literatur für die Alpha-Beta-Suche zur überprüfen und den Monte Carlo Tree Search-Algorithmus umzusetzen. Für beide Algorithmen werden weitere Techniken entwickelt und zunächst für jeden Algorithmus einzeln systematisch verglichen. Abschließend erfolgt ein direkter Vergleich der beiden Methoden.

Zunächst wird in Kapitel 2 das Spiel Abalone vorgestellt und kurz analysiert. Danach werden in Kapitel 3 die beiden Algorithmen und deren Weiterentwicklungen eingeführt. Nachdem die Durchführung der Tests erläutert wurde (siehe Kapitel 4), werden dessen Ergebnisse in Kapitel 5 präsentiert. Abschließend werden diese im Kapitel 6 diskutiert, bevor schließlich ein Fazit gezogen wird (siehe Kapitel 7).

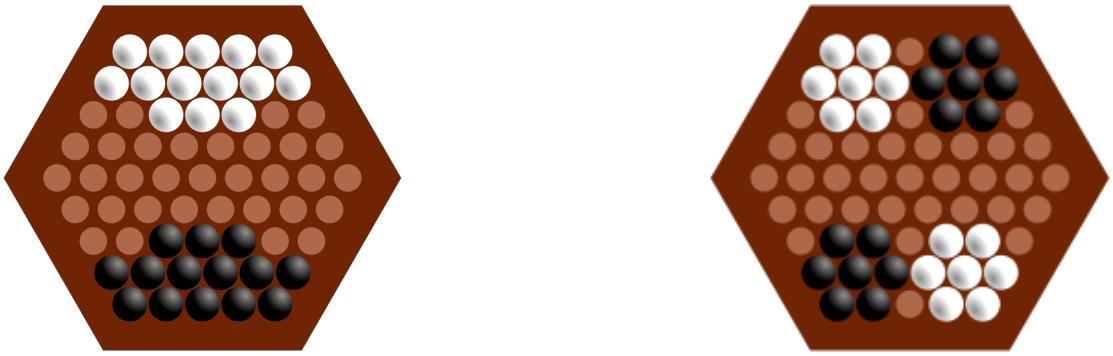
[1] Zur besseren Lesbarkeit wird in dieser Bachelorarbeit das generische Maskulinum verwendet. Die in dieser Arbeit verwendeten Personenbezeichnungen beziehen sich - sofern nicht anders kenntlich gemacht – auf alle Geschlechter.

2. Das Spiel Abalone

In diesem Kapitel wird eine kurze Übersicht über das in der Arbeit behandelte Spiel gegeben. Es werden die Spielregeln erklärt und die Spieltheorie sowie die technische Umsetzung betrachtet.

2.1. Spielregeln

Das Spiel Abalone ist ein strategisches Zwei-Spieler-Brettspiel, welches auf einem hexagonalem Feld mit 61 Positionen gespielt wird. Es gibt jeweils 14 weiße und schwarze Kugeln. Die Farbe Schwarz beginnt das Spiel. Als Startaufstellungen gibt es unter anderem die Standard- (siehe Abb. 2.1a) und die Belgium-Daisy-Startaufstellung (siehe Abb. 2.1b), die zu einem schnelleren und aktiveren Spielverlauf führen kann. Das Ziel ist es, sechs gegnerische Kugeln vom Spielfeld zu drängen (im Folgenden auch „schlagen “ genannt).



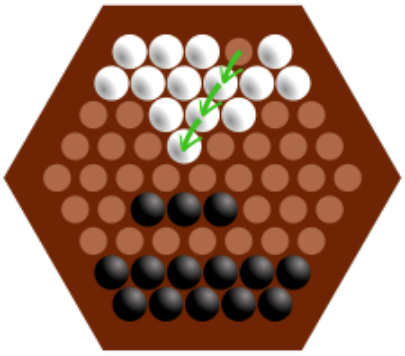
(a) Standard (b) Belgium-Daisy

Abbildung 2.1.: Startaufstellungen [21]

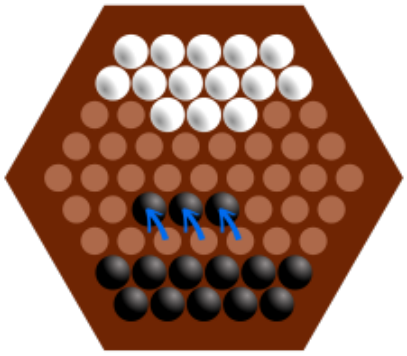
Insbesondere in Wettkämpfen wird das Spiel oft zusätzlich mit einer Zeitbegrenzung von 10 bis 15 Minuten pro Farbe durchgeführt. [13]

2.2. Spielzüge

Bei einem normalen Zug kann der Spieler bis zu drei seiner Kugeln bewegen. Eine Kugel kann in alle ihre benachbarten Positionen bewegt werden. Wenn mehrere Kugeln geschoben werden, müssen diese zusammenhängend in einer Linie liegen und in dieselbe Richtung bewegt werden. Jede Kugel darf nur eine Position weiter geschoben werden. Die beiden daraus resultierenden möglichen Zugarten werden als Linien- (siehe Abb. 2.2a) und Seitwärtszug (siehe Abb. 2.2b) bezeichnet.



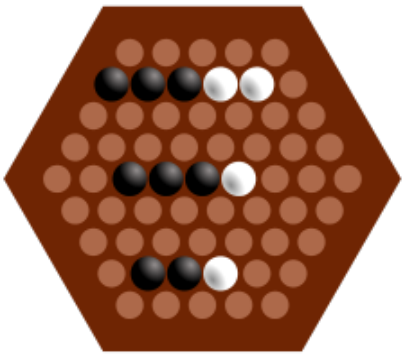
(a) Linienzug



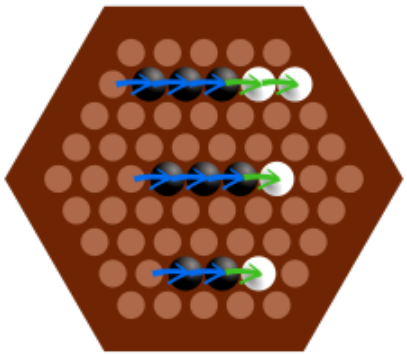
(b) Seitwärtszug

Abbildung 2.2.: Zugbewegung [20]

Mithilfe der sogenannten „Sumito-Stellung“ können die gegnerischen Kugeln in das freie Feld hinter ihnen oder vom Spielfeld gedrängt werden. Die Anzahl der eigenen bewegten Kugeln muss dabei größer sein als die der gegnerischen bewegten Kugeln. In Abbildung 2.3a sind die möglichen Sumito-Stellungen gezeigt und Abbildung 2.3b stellt die Situation nach Ausführung des jeweiligen Zuges dar. [13]



(a) Sumito-Stellung



(b) Stellung nach Zugausführung

Abbildung 2.3.: *Sumito*-Stellung. Schwarz ist am Zug.[20]

2.3. Spieltheorie

Abalone gehört zu der Kategorie der kombinatorischen Spiele. Diese erfüllen die folgenden Eigenschaften:

- Null-Summen-Spiel: Aus dem Sieg des einen Spielers folgt die Niederlage des anderen Spielers.

- Perfekt informiertes Spiel: Zu jedem Zeitpunkt des Spiels können die Spieler jede theoretisch nachfolgende Situation voraussehen, da alle Information vorhanden sind. Es gibt keine versteckten Informationen.
- Deterministisches Spiel: Jede Aktion (Zug) ist klar berechenbar. Es gibt keinen Zufall.
- Sequentielles Spiel: Einzelne Aktionen werden sequentiell (also nacheinander) ausgeführt und keine Aktionen verlaufen simultan.
- Diskretes Spiel: Alle Aktionen sind diskret. Es gibt keine kontinuierlichen Aktionen.[6]

Der durchschnittliche Verzweigungsgrad, also die durchschnittliche Anzahl der möglichen Spielzüge für eine Spielsituation, beträgt 60. Im Schnitt benötigt es 87 Halbzüge (Ausführung eines Spielzug durch einen der Spieler) bis ein Spiel beendet ist.[14]

2.4. Technische Umsetzung des Spiels

Das Spielfeld wird im Programm als eindimensionales *Array* dargestellt. Weiterhin werden jeweils die Positionen der schwarzen und weißen Kugeln sowie Informationen über den Spielstand, Spieler, Spielverlauf und der Hashwert (siehe Kapitel 3.1.4) gespeichert.
Ein Spielzug besteht aus den betroffenen Spielfeldpositionen sowie Informationen zu Zugrichtung und -art.
Bei der Berechnung der Spielzüge wird durch die Kugelpositionen der betroffenen Farbe iteriert. Dabei werden für jede Kugel die möglichen Züge berechnet. Eine Abspeicherung und somit eine dynamische Wiederverwendung der bereits berechneten Züge aus vorherigen Spielsituationen findet nicht statt. Da insbesondere gute Züge bis zu fünf Spielkugeln (drei eigene und zwei gegnerische) bewegen und dadurch auch sämtliche Positionen und Kugeln im Umkreis von fünf Feldern betroffen sein können, wurde sich gegen eine dynamische Programmierung des Spielzuggenerators entschieden.
Das Spielfeld wird anfangs als *String* übergeben. Dabei wird für die schwarzen Kugeln eine kleines und für die weißen Kugel ein großes K verwendet. Freie Positionen werden mit einer 1 dargestellt. Weiterhin beinhaltet der *String* am Ende die Anzahl der gespielten Halbzüge und die Farbe, die als nächstes am Zug ist (w für Weiß/ *White* und b für Schwarz/ *Black*). Für die beiden Startaufstellungen sehen die *Strings* somit wie folgt aus:

- Standardaufstellung:
  kkkkkkkkkkk11kkk1111111111111111111111111111KKK11KKKKKKKKKKKb 0
- Belgium-Daisy-Aufstellung:
  kk1KKkkkKKK1kk1KK1111111111111111111111111111KK1kk1KKKkkkKK1kkb 0

3. Algorithmen

Im Folgenden werden die beiden zu untersuchenden Algorithmen und ihre Weiterentwicklungen beschrieben.

3.1. Alpha-Beta-Suche

Zunächst wird die Alpha-Beta-Suche vorgestellt. Diese basiert auf einem heuristischen Ansatz.

3.1.1. Basis-Algorithmus

Die Alpha-Beta-Suche (im Folgenden auch Alpha-Beta-Algorithmus genannt) ist ein tiefenorientiertes Suchverfahren, das auf dem Minimax-Verfahren basiert und besonders für Zwei-Spieler-Spiele geeignet ist.
Beim Minimax-Verfahren wird der Spielbaum rekursiv durchlaufen. Auf jeder Ebene wird zwischen der Maximierung und Minimierung gewechselt. Ein Spieler maximiert auf seiner Ebene und führt einen Zug aus, während der andere Spieler auf der nächsten Ebene versucht, den Spielwert zu minimieren. Der Spieler an der Wurzel des Baumes, also die letztlich zu untersuchende Spielstellung, maximiert grundsätzlich den Spielwert.
Auf diese Weise entsteht ein Suchbaum mit allen möglichen Spielzugkombinationen. In der Regel wird die Suche bei einer bestimmten Tiefe abgebrochen und der Spielwert wird mithilfe einer Bewertungsfunktion (siehe Kapitel 3.1.2) ermittelt.
Es wird eine optimale Spielweise von beiden Spielern angenommen.[3]

Die Alpha-Beta-Suche optimiert das Minimax-Suchverfahren, indem angenommen wird, dass viele Spielzüge ohne Informationsverlust ignoriert werden können. Dazu wird ein Intervall $[\alpha, \beta]$ eingeführt, das anfangs mit $[-\infty, \infty]$ initialisiert und während der Suche aktualisiert wird. Wenn der maximierende Spieler auf seiner Ebene einen Wert $> \alpha$ findet, wird α auf diesen Wert gesetzt, da der maximierende Spieler somit einen Zug gefunden hat, der mindestens diesen Spielwert bringt. Wenn der minimierende Spieler wiederum auf seiner Ebene einen Wert $< \beta$ findet, wird β angepasst.
Zu einem sogenannten *Cutoff*, also dem nicht weiteren Betrachten von Teilbäumen (bzw. Zügen), kommt es zum Beispiel, wenn der maximierende Spieler einen Zug mit einen Wert $\geq \beta$ findet (β-*Cutoff*). In diesem Fall würde der minimierende Spieler es nicht zu dieser Spielsituation kommen lassen, da es für ihn bereits einen besseren Spielzug gibt. Gleiches gilt, wenn der minimierende Spieler einen Wert $\leq \alpha$ findet (α-*Cutoff*).

Die Laufzeit des Minimax-Suchverfahrens ist $\Theta(k^h)$, wobei h die Höhe des Suchbaumes (Anzahl der Spielzüge) und k der Verzweigungsgrad (mögliche Spielzüge pro Ebene) sind. Die Alpha-Beta-Suche hat im schlechtesten Fall die gleiche Laufzeit. Mit einer guten Zugsortierung (siehe Kapitel 3.1.5) und

somit einem effizienten Abschneiden von Knoten kann die Alpha-Beta-Suche jedoch im besten Fall zu einer Laufzeit von $\Theta(k^{h/2})$ führen. [3]

Die Implementierung des Algorithmus kann unter der Annahme, dass jeder Spieler aus seiner Sicht versucht den Spielwert zu maximieren, kompakt erfolgen (siehe Pseudocode 1 [4]). Dabei muss beachtet werden, dass der Rückgabewert des rekursiven Aufrufs negiert werden muss, da jeder Spieler maximiert. Außerdem müssen Intervallgrenzen α und β bei der Rekursion getauscht und negiert werden (Zeile 9).
Die Auswertung der Spielsituation, wenn die maximale Suchtiefe oder eine Terminalstellung erreicht ist, findet in Zeile 3 statt. In Zeile 13 kommt es gegebenenfalls zu einem *Cutoff*. [4]

Algorithm 1 Negamax

```
1:  function NEGAMAX(board, depth, alpha, beta)
2:      if depth = 0 or board is terminal then
3:          return Evaluate(board)
4:      end if
5:      maxValue ← −∞
6:      for all move in GenerateMoves(board) do
7:          alpha ← max(alpha, maxValue)
8:          makeMove(move, board)
9:          value ← −Negamax(board, depth − 1, −beta, −alpha)
10:         undoMove(move, board)
11:         if value > maxValue then
12:             maxValue ← value
13:             if maxValue ≥ beta then
14:                 break
15:             end if
16:         end if
17:     end for
18:     return maxValue
19: end function
```

3.1.2. Bewertungsfunktion

Die Bewertungsfunktion ist ein zentrales Element der Alpha-Beta-Suche und entscheidend dafür, wie zielführend der gefundene Zug letztendlich ist. In den meisten Fällen wird die Alpha-Beta-Suche nicht bis zum Erreichen einer Terminalstellung des Spiels durchgeführt. Die Bewertungsfunktion versucht daher, den Spielwert anhand bestimmter Kriterien möglichst genau zu schätzen. Dieser Spielwert gibt an, wie wahrscheinlich es ist, dass der Spieler das Spiel gewinnen wird. Spielsituationen, in denen ein Sieg für den Spieler wahrscheinlicher ist, werden positiv evaluiert. Neutrale Spielsituationen werden mit null und Spielsituationen, die eher zu eine Niederlage führen, negativ gewertet.

Die Erstellung einer Bewertungsfunktion erfordert in der Regel ein hohes Maß an Domänenwissen, wie beispielsweise Kenntnisse über Spielstrategien oder besondere Spielstellungen. Grundsätzlich

unterscheidet man zwischen Materialkriterien und Raum- bzw. Strategiekriterien. Im Fall des Spiels Abalone ist als Material nur die Anzahl der Kugeln zu berücksichtigen, während die Raum- und Strategiekriterien vielfältiger sind. Zu bedenken ist, dass alle Kriterien letztendlich nur Abschätzungen über die Stärke der Spielsituation sind und unter bestimmten Umständen zu Fehlinterpretationen führen können.[19]
Die hier benutzte Bewertungsfunktion orientiert sich an der bereits ausgearbeiteten Version aus der Arbeit „Implementing a Computer Player for Abalone using Alpha-Beta and Monte-Carlo Search“ von Pascal Chorus [16] und führt kleine Ergänzungen und Veränderungen ein. Die Funktion besteht aus den folgenden Kriterien:

- **Sieg oder Niederlage:** Bei der Bewertung der Spielsituation wird einem Sieg ein hoher positiver Wert zugewiesen, während eine Niederlage mit einem ebenso hohen negativen Wert bewertet wird. Diese Werte sind bewusst hoch gewählt, um sicherzustellen, dass andere Kriterien keine Rolle mehr bei der Beurteilung dieser Situationen spielen. Wenn das Spiel noch nicht entschieden ist, wird dieses Kriterium mit 0 gewertet.

- **Anzahl der Kugeln:** Die Anzahl der eigenen und gegnerischen Kugeln ist maßgeblich für die Gewinnchancen verantwortlich. Zu einem dürfen nicht zu viele eigene Kugeln geschlagen werden, da ansonsten das Spiel verloren wird. Zum anderen ist es leichter den Gegner zu besiegen, je größer die Differenz zwischen der Anzahl der eigenen und gegnerischen Kugeln ist. Daher wird in der Funktion die Differenz der Kugelanzahlen berechnet. Ein negatives Ergebnis bedeutet, dass der Gegner mehr Kugeln hat, während ein positives Ergebnis darauf hinweist, dass der Spieler selbst mehr Kugeln besitzt.

- **Entfernung zum Zentrum:** Bei Abalone ist es von Vorteil, wenn die eigenen Kugeln die Positionen in der Spielfeldmitte besetzen. Zu einem können Kugeln, die in der Mitte liegen, nicht so einfach vom Spielfeld geschoben werden. Zum anderen können Kugeln des Gegners von der Mitte aus effektiver und mit weniger Risiko zum Rand gedrängt werden. Als Maß für die Berechnung der Distanz einer Kugel zur Mitte wird die Anzahl der Züge m berechnet, die es benötigt, um in die Mitte zu kommen. Die Bewertung erfolgt anhand der Differenz zwischen der maximalen Entfernung (Wert 4) und m. Somit erhält die Kugel in der Mitte des Feldes die höchste Wertung (4). Die Werte aller Kugeln einer Farbe werden aufsummiert. Anders als in der Arbeit von Pascal Chorus wird dieser Wert mithilfe der aktuellen Anzahl der Kugeln einer Farbe normiert, um somit auch hohe Werte dieses Kriteriums bei Spielsitutationen mit weniger Kugeln zu ermöglichen. Die Entfernung zum Zentrum für die gegnerischen Kugeln fließt negativ in den Spielwert mit ein.

- **Kompaktheit:** Ein weiterer entscheidender Faktor ist die Kompaktheit der eigenen Kugeln. Durch enge Formationen können einzelne Kugeln schwieriger angegriffen werden und eigene Angriffe sind effizienter. Um dieses Kriterium zu berechnen, wird für jede Kugel die Anzahl von Nachbarkugeln der eigenen Farbe berechnet (Werte zwischen 0 und 6) und anschließend je Farbe die Summe aller Kugelwerte gebildet. Wieder wird im Gegensatz zur genannten Arbeit der Wert normiert. Dies führt zu Werten zwischen 0 und 4,14, da es keine Position gibt, in der alle Kugeln von sechs Kugeln der eigenen Farbe umgeben sein können. Die Kompaktheit der gegnerischen Kugeln wird wieder negativ gewichtet.

- **Angriff:** Um den Gegner besiegen zu können, müssen erst einmal möglichst viele Angriffssituationen bzw. *Sumito*-Stellungen erzeugt werden. In der Arbeit von Pascal Chorus werden diese Situationen für beide Seiten einfach gezählt (1 Punkt pro Angriff). In dieser Bewertungsfunktion wird zusätzlich differenziert, ob nur eine gegnerische Kugel (1 Punkt), zwei gegnerische Kugeln (2 Punkte) oder sogar eine Kugel vom Rand (8 Punkte) geschoben werden kann. Zudem erfolgt die Auswertung dieses Kriteriums auf Grund des Rechenaufwandes erst nach 20 Halbzügen. Der Angriffswert der Gegners wird negativ gewertet. [16]

- **Formation durchbrechen:** Als zusätzliches Kriterium im Vergleich zur originalen Bewertungsfunktion aus der erwähnten Arbeit wird analysiert, wie oft Gruppen des Gegners durch eine eigene Kugel unterbrochen sind. Dies bietet den Vorteil, dass die Kugeln des Gegners getrennt werden und die eigenen Kugeln in der Regel nicht von den gegnerischen Kugeln geschoben werden können, da diese sich gegenseitig blockieren. Zur Berechnung wird gezählt, wie oft eine eigene Kugel zwischen zwei gegnerischen Kugeln in einer Linie liegt. Somit kann für eine Kugel der Wert zwischen null und drei liegen. Alle Werte der Kugeln werden aufsummiert. Der Wert des Gegners wird negativ gewertet. [14]

Neben den Kriterien selbst ist auch die Gewichtung entscheidend für die Stärke der Bewertungsfunktion. Die Gewichtung bestimmt, wie relevant jedes einzelne Kriterium ist. Zum Beispiel ist die Anzahl der Kugeln ein besonders entscheidendes Kriterium für den Ausgang der Spielsituation, während die Anzahl der durchbrochenen Formationen möglicherweise nicht so stark über Sieg oder Niederlage eines Spiels entscheidet. Die Tabelle 3.1 zeigt die verwendete Basisgewichtung.

Kriterium	**Gewichtung**
Sieg oder Niederlage	± 1.000.000
Anzahl der Kugel	1.000
Entfernung zum Zentrum	130
Kompaktheit	70
Angriff	30
Formationen durchbrechen	10

Tabelle 3.1.: Basisgewichtung der Bewertungsfunktion

Diese Basiswerte werden noch einmal zusätzlich gewichtet, um offensivere oder defensivere Spieler zu simulieren. Die möglichen zusätzlichen Gewichtungen sind die folgenden:

- gegnerische (gegn.) Kugeln
- eigene (eig.) Kugeln
- Distanz zum Zentrum
- eig. Kompaktheit
- gegn. Kompaktheit
- eig. Angriffe
- gegn. Angriffe
- eig. durchbrochene Formationen
- gegn. durchbrochene Formationen

Offensivere Spieler legen möglicherweise einen höheren Fokus auf die eigenen Angriffe und das Besiegen von gegnerischen Kugeln, auch unter Verlust der eigenen Kugeln. Dagegen achten defensivere Spieler zum Beispiel eher auf eine eigene hohe Kompaktheit, geringe Entfernung zum Zentrum und

auf das Bewahren der eigenen Kugeln.[16]
Welche Spielweise bzw. Gewichtung stärker ist, wird in den Experimenten ermittelt.

Für Verlgeichszwecke wird ebenfalls die Bewertungsfunktion aus der Arbeit von Pascal Chorus umgesetzt. Diese wird mit der hier entwickelten Funktion verglichen. Da keine genauen Implementationsdetails bekannt sind, wird nur die funktionale und konzeptionelle Ebene getestet. Wichtige Aspekte wie der Rechenaufwand können nicht direkt thematisiert werden.

3.1.3. Iterative Tiefensuche

Im Anwendungsbereich der Alpha-Beta-Suche ist es unpraktisch, die Suchtiefe im Voraus festzulegen, da oft nur eine begrenzte Zeit für die Suche zur Verfügung steht. Dadurch besteht bei einer festen Tiefe das Risiko, dass die Suche noch nicht abgeschlossen ist und daher kein oder kein optimales Ergebnis zurückgegeben werden kann. Um dieses Problem zu umgehen, wird die iterative Tiefensuche eingeführt. Dabei wird der Suchalgorithmus wiederholt mit zunehmender Suchtiefe aufgerufen. Wenn eine Suche bei einer Tiefe zeitlich nicht mehr abgeschlossen werden kann, wird das Ergebnis aus der vorherigen Iteration verwendet.

Zunächst mag diese Technik ineffektiv erscheinen und zu Mehraufwand in der Suche führen. Jedoch können Informationen aus vorherigen Iterationen verwendet werden, um Techniken wie das *Aspiration Window* (siehe Kapitel 3.1.6) oder die Zugsortierung (siehe Kapitel 3.1.5) zu verbessern. [19]
Darüber hinaus ist der Mehraufwand bei Entscheidungsproblemen mit einem hohen Verzweigungsgrad, wie er bei Abalone gegeben ist (siehe Kapitel 2.3), im Verhältnis gering. Dies liegt daran, dass die meiste Suchzeit für die jeweils tiefste Suchebene verwendet wird und somit höhere Suchebenen bzw. vorherige Iterationen keinen großen Beitrag leisten.[12]

3.1.4. Transposition Table

In Spielen wie Abalone, die einen symmetrischen Aufbau haben und in denen nur eine Art von Spielstein vorhanden ist, treten oft wiederkehrende Spielsituationen auf. Diese Situationen können auf verschiedenen Pfaden bzw. durch verschiedene Zugkombinationen erreicht werden.
Um die Berechnung solcher Situationen im Suchbaum zu optimieren, werden bereits untersuchte Spielsituationen in einer Tabelle gespeichert. Die Tabelle wäre allerdings zu groß um für jede Spielsituation einen einzelnen Eintrag zu haben. Ebenfalls wäre das Finden eines Eintrages zeitaufwendig. Um diese Probleme zu umgehen wird eine *Transposition Table* eingeführt.

Für die Verwendung einer *Transposition Table* wird zunächst eine Methode benötigt, die jeder Spielsituation einen möglichst einzigartigen Wert (Hashwert) zuweist. Das Zobrist-Hashing erweist sich bei vielen Spielen als eine effiziente Hashwertberechnung. Hierbei werden jeder Position auf dem Spielbrett zwei 64-Bit-Zahlen zugewiesen: eine für die Farbe Schwarz und eine für die Farbe Weiß. Zusätzlich wird eine weitere 64-Bit-Zahl verwendet, um den Spieler anzugeben, der am Zug ist. Die Berechnung des Hashwertes erfolgt, indem für jede Kugelposition die zur Farbe passende Zahl gewählt wird. Die Zahlen aller Kugeln und die Spielerzahl werden dann mithilfe der XOR-Operation zusammengerechnet. [4]
Der entstandene Wert ist nahezu einzigartig. Bei einer Suche mit 10.000.000 Spielsituationen ist die

Wahrscheinlichkeit, dass es zwei verschiedene Spielsituationen mit dem gleichen Hashwert gibt, mit $2,71 \times 10^{-6}$ sehr gering.[16]

Die XOR-Operation bietet mehrere Vorteile: Zum einem ist es eine schnelle bitweise Operation. Zudem kann der Hashwert inkrementell entstehen, in dem nach und nach jede Zahl der Kugelpositionen mit dem bereits bestehenden Hashwert durch die XOR-Operation verrechnet wird. Zuletzt ist es durch die selbstinverse Operation XOR nicht nötig, für jede neue Spielsituation einen neuen Hashwert zu berechnen. Vielmehr können mit der XOR- Operation alte Informationen (z.B. die alte Kugelposition) gelöscht werden und neue Informationen (z.B. die neue Kugelposition) hinzugefügt werden.[4]

Um jedem Hashwert nun einen eigenen Eintrag in der Tabelle zu ermöglichen, bedarf es einer Größe von 2^{64} Einträgen und somit einer zu großen Tabelle. Daher wird eine kleinere Tabelle mit 2^{20} Einträgen genutzt. Dabei werden die letzten 20 Bits des Hashwerts verwendet, um den Eintrag in der Tabelle zu bestimmen, während der komplette 64-Bit-Schlüssel mitgespeichert wird, um die Spielsituation eindeutig zu identifizieren. Dadurch kann es zu Kollision der Einträgen kommen, da zwei Hashwerte auf den gleichen Tabelleneintrag projeziert werden.[15]
Mit einer solchen Situation kann auf verschiedene Weise umgegangen werden. In dieser Arbeit wird überprüft, welcher der beiden Einträge den tieferen Teilsuchbaum hat. Hierbei wird angenommen, dass bei einem Eintrag mit einem größeren Teilsuchbaum auch mehr Knoten untersucht wurden und somit die Zeitersparnis bei der Wiederverwendung dieses Eintrages am größten ist.[5]

Ein Eintrag in der *Transposition Table* besteht insgesamt aus den folgenden Informationen:

- Hashwert: Der gesamte 64-Bit Hashwert wird für die eindeutige Identifikation des Spielsbretts gespeichert.
- Spielwert: Der Wert entspricht dem bisher für diese Spielsituation gefundenen Spielwert. Er kann ein exakter Wert sein oder aber auch nur eine untere bzw. obere Schranke bilden.
- *Flag*: Es wird angegeben, ob es sich bei dem Spielwert um einen exakten Wert oder um eine Schranke handelt.
- Tiefe des Teilsuchbaums: Es wird angegeben, wie viele Ebenen von dieser Position aus durchsucht wurden, um den Spielwert zu ermitteln.
- Bester Spielzug: Dies entspricht dem bisher bestem gefundenen Zug für diese Position. Dieser kann für die Zugsortierung (siehe Kapitel 3.1.5) verwendet werden.

Der Pseudocode 2 zeigt die Verwendung der *Transposition Table* in der Alpha-Beta-Suche.[4] In Zeile 6 wird geprüft, ob es für eine Spielsituation bereits einen Eintrag gibt. Anschließend muss bei einem gefundenen Eintrag überpüft werden, ob von der aktuellen Position weniger oder gleich tief gesucht wird, wie von der in der Tabelle gespeicherten Spielsituation (Tiefe des Teilsuchbaumes). Wenn dies nicht der Fall ist, kann der Spielwert des Eintrages nicht verwendet werden, da der gespeicherte Wert durch eine weniger tiefe Suche ermittelt wurde und es somit zu Informationsverlusten kommen kann. Aus diesem Grund können auch keine Einträge aus vorherigen Iterationen der iterativen Tiefensuche verwendet, sondern müssen immer ersetzt werden. Dennoch kann für die Zugsortierung der beste bisher gefundene Zug verwendet werden (siehe Zeile 8). Insgesamt ist diese Bedingung der Gültigkeit

eines Eintrages ein weiteres Argument für die Strategie, die Tabelleneinträge anhand der Tiefe des Teilsuchbaumes zu ersetzen.
Wenn der Eintrag verwendet werden kann, muss weiterhin überprüft werden, ob der gespeicherte Wert ein exakter Wert oder nur die obere Schranke β bzw. untere Schranke α ist (siehe Zeile 9 - 15). Je nach Situation kann der Wert direkt zurückgegeben, das Suchfenster verkleinert oder ein Cutoff erzeugt werden.
In den Zeilen 22 bis 28 werden die Informationen der aktuellen Spielsituation in der *Transposition Table* gespeichert.[4]

Algorithm 2 Negamax with Transposition Table

```
 1: function NEGAMAX(board, depth, alpha, beta)
 2:     if state is terminal or depth = 0 then
 3:         return Evaluate(state)
 4:     end if
 5:     hash ← ComputeHash(state)
 6:     if hash is found in Transposition Table then
 7:         entry ← Transposition Table[hash]
 8:         bestMove ← entry[bestMove]
 9:         if entry.depth ≥ depth then
10:             if entry.flag = EXACT then
11:                 return entry.value
12:             else if entry.flag = LOWER and entry.value > alpha then
13:                 alpha ← entry.value
14:             else if entry.flag = UPPER and entry.value < beta then
15:                 beta ← entry.value
16:             end if
17:             if alpha ≥ beta then
18:                 return entry.value
19:             end if
20:         end if
21:     end if
22:     Search                                          ▷ negamax Alogrithm
23:     flag ← EXACT
24:     if bestValue ≤ alpha then
25:         flag ← UPPER
26:     else if bestValue ≥ beta then
27:         flag ← LOWER
28:     end if
29:     UpdateTranspositionTable(hash, depth, bestValue, flag)
30:     return bestValue
31: end function
```

Die Tabelle wird jedes mal beim Starten des Suchalgorithmus neu initialisiert und nicht über das gesamte Spiel global verwaltet.

3.1.5. Zugsortierung

Wie bereits im Kapitel 3.1.1 erwähnt, ist die Zugsortierung besonders wichtig, um viele Cutoffs generieren zu können. Die Sortierung von Zügen erfordert zwar zusätzlichen Zeitaufwand, jedoch müssen in der Regel auch deutlich weniger Spielsituationen untersucht werden. Gerade bei Spielen mit einem hohen Verzweigungsgrad ist es wichtig, nicht alle Züge untersuchen zu müssen. Im optimalen Fall wird immer zuerst der beste Zug betrachtet. Dadurch kann es zu der bereits erwähnten *best-case* Laufzeit von $\Theta(k^{h/2})$ kommen, wobei h die Höhe des Suchbaumes und k der Verzweigungsgrad sind. Grundsätzlich ist jedoch unklar, welcher Zug der beste ist, weshalb es verschiedene Techniken für eine annähernd optimale Sortierung gibt.

Als Basis für die Zugsortierung in Abalone werden verschiedene Eigenschaften der Züge betrachtet. Besonders gute Züge sind solche, die angreifen oder sogar gegnerische Kugel vom Spielfeld schieben. Ebenfalls werden Züge bevorzugt, die mehrere Kugeln bewegen, da dadurch die Kugeln zusammenbleiben. Linienzüge werden höher bewertet als Seitwärtszüge, da nur Linienzüge angreifen können. Anhand dieser Kriterien können verschiedene Zugreihenfolgen gebildet werden. In der Arbeit von Pascal Chorus wurden bereits verschiedene Ordnungen untersucht und folgende als erfolgreichste bewertet:

1. 3 Kugeln, Schlagen
2. 2 Kugeln, Schlagen
3. 3 Kugeln, Angriff
4. 2 Kugeln, Angriff
5. 3 Kugeln, Linienzug
6. 2 Kugeln, Linienzug
7. 3 Kugeln, Seitwärtszug
8. 2 Kugeln, Seitwärtszug
9. 1 Kugel

Diese Zugsortierung wird als Basissortierung genutzt.[16]

Eine besonders effektive und genaue Methode zur Zugsortierung ist die Verwendung der Bewertungsfunktion. Dabei wird jeder Zug und die daraus resultierende neue Spielsituation anhand der Bewertungsfunktion evaluiert und entsprechend sortiert.
Die Sortierung ist aufgrund der Bewertung der Spielsituationen aufwendiger, aber sollte auch zu größeren *Cutoffs* und somit einer signifikanteren Minimierung des Suchaufwandes führen. Insbesondere auf höheren Ebenen bringt eine gute Zugsortierung eine große Reduktion der zu untersuchenden Knoten (Spielsituationen) mit sich. Auf tieferen Ebenen müssen immer mehr Züge untersucht und somit

auch sortiert werden, weshalb die Sortierung weniger Rechenaufwand benötigen sollte. Daher wird die Zugsortierung mit der Bewertungsfunktion nur auf den ersten drei Ebenen durchgeführt. Auf den folgenden Ebenen wird wieder die Basissortierung verwendet.

Um die Basiszugsortierung noch zu verbessern, wird sich die Technik der iterativen Tiefensuche und der *Transposition Table* zu Hilfe genommen. Zunächst werden für jede Suchebene zwei Züge (*Killer-Moves*), die einen *Cutoff* auf dieser Ebene erzeugt haben, gespeichert. Diese werden dann an anderer Stelle auf derselben Ebene, sofern es dort zulässige Züge sind, zuerst untersucht. Hierbei wird angenommen, dass Züge, die bereits einen *Cutoff* bei einer ähnlichen Spielsituation hervorgerufen haben, dies wahrscheinlich wieder tun werden.
Zuletzt wird in der *Transposition Table* jeweils der beste ermittelte Zug (*Hash-Move*) für eine Spielsituation gespeichert. Dieser Zug wird bei einer wiederkommenden Spielsituation zuallererst untersucht.
Die *Killer-Moves* und der *Hash-Move* können aus den vorherigen Iterationen der Tiefensuche wiederverwendet werden, wodurch die iterative Tiefensuche hier einen großen Vorteil bieten kann. [15]

3.1.6. Aspiration Window

Durch die iterative Tiefensuche (siehe 3.1.3) ist der beste mögliche Wert aus einer niedrigeren Suchtiefe bereits bekannt. Mit dieser Information und der Annahme, dass sich der Wert durch eine weitere Suchebene nicht stark verändern sollte, wird bei der Technik des *Aspiration Windows* versucht, die Alpha-Beta-Suche mit einem kleineren Fenster als $[-\infty, \infty]$ zu starten. Stattdessen wird die Suche mit einem Fenster um den letzten Wert ($[lastValue - offset, lastValue + offset]$) begonnen. Die Fenstergröße ist somit zweimal der Wert des Offsets ($FG = 2 * Offset$). Eine passende Größe muss experimentell ermittel werden.
Bei der Technik kann es durch das kleinere Fenster beim Start des Algorithmus zur schnelleren Suche mit größeren *Cutoffs* kommen. Allerdings kann es auch vorkommen, dass die Suche wiederholt werden muss, da der tatsächlich beste Wert außerhalb des Fensters liegt. Wenn der ermittelte Wert $\leq \alpha$ ist, muss die Suche mit einem Fenster $[-\inf, value]$ neu gestartet werden. Ebenso muss die Suche mit einem Fenster von $[value, \inf]$ wiederholt werden, wenn der gefundene Wert $\geq \beta$ ist. [10]

3.1.7. Ruhesuche

In vielen Spielen, insbesondere in Abalone, kann es zu dem Problem des Horizonteffekts kommen. Dieser kann auftreten, wenn eine gegnerische Kugel vom Brett gedrängt wird. Auf den ersten Blick mag dies vorteilhaft erscheinen, bringt aber möglicherweise auch eigene Kugeln in Gefahr, indem diese näher an den Rand des Spielbretts geraten. Der Algorithmus bewertet einen solchen Zug auf der letzten Ebene als vielversprechend, da er eine gegnerische Kugel vom Brett drängt und dadurch das Spielfeld positiv beeinflusst. Jedoch kann in der nächsten Runde genau dieser Zug dazu führen, dass eine eigene Kugel vom Gegner eliminiert wird. Dies wird vom Algorithmus jedoch nicht erkannt, da die Suche vorher beendet wurde.
Um den Horizonteffekt zu vermeiden, wird die Ruhesuche angewendet. Bei der Ruhesuche wird die Suche fortgesetzt, wenn der letzte Zug eine „Schlag“-Aktion war, also eine Kugel vom Feld gedrängt wurde. Die Suche wird erst dann beendet, wenn sich die Situation beruhigt hat, also keine weiteren Schlagzüge in Folge mehr möglich sind. Es werden ausschließlich Schlagzüge in der Ruhesuche

betrachtet. [8]
Diese Technik führt zwar potenziell zu einem erhöhten Suchaufwand, da gewisse Blätter im Baum weiteruntersucht werden, verhindert aber die Fehlinterpretation von Spielzügen und kann somit zu einer stärkeren Spielweise führen. Dies wird experimentell geprüft.

3.2. Monte Carlo Tree Search (MCTS)

Im folgenden wird der Monte Carlo Tree Search-Algorithmus vorgestellt. Dieser basiert auf einem stochastischen Ansatz.

3.2.1. Basis-Algorithmus

Der Monte Carlo Tree Search-Algorithmus (MCTS-Algorithmus) ist ein Verfahren der Bestensuche, das in einem iterativen Prozess einen asymmetrischen Suchbaum aufbaut, solange das Zeitbudget nicht erschöpft ist. Damit agiert der MCTS-Algorithmus wie der Alpha-Beta-Algorithmus zwar auf einem Suchbaum. Allerdings wird die Ermittlung des besten Spielzugs durch Simulationen und Wahrscheinlichkeiten bestimmt und ist somit grundlegend anders als der heuristische Ansatz der Alpha-Beta-Suche.

Jeder Knoten im Suchbaum repräsentiert eine Spielsituation und speichert zusätzlich Informationen wie die Anzahl der Besuche dieses Knotens und den von ihm ausgehenden Gewinnwert. Weiterhin wird der Elternknoten sowie der Spielzug, der zu dieser Spielsituation geführt hat, gespeichert. Der Pseudocode 3 illustriert den Ablauf des Algorithmus.[6]

Algorithm 3 Monte Carlo Tree Search (MCTS)

```
1:  function MCTS(root)
2:      while time is not over do
3:          node ← root
4:          while node has children do
5:              node ← selectChild(node)
6:          end while
7:          if node is not terminal and node is visited then
8:              expandNode(node)
9:              node ← selectRandomNode(node.children)
10:         end if
11:         result ← simulation(node)
12:         backpropagate(node, result)
13:     end while
14:     return selectBestChild(root)
15: end function
```

Jede Iteration besteht dabei aus den vier folgenden Phasen:

1. **Selektion** (Zeile 4-6): In jeder Iteration beginnt der Algorithmus am Wurzelknoten des Baumes, der die aktuelle Spielsituation repräsentiert. Von dort aus wird der Baum bis zu einem expandierbaren oder noch nicht besuchten Knoten durchlaufen. Das Durchlaufen geschieht mithilfe der *Tree Policy* (siehe Kapitel 3.2.2), die den nächsten Kindknoten auswählt. Ein expandierbarer Knoten ist ein Knoten, der keine Endspielstellung darstellt und für den noch keine Kindknoten generiert wurden.

2. **Expandierung** (Zeile 7-10): Bei der Expandierung werden zunächst alle möglichen Folgezustände einer Spielsituation generiert. Die Anzahl der Simulationen und die Ergebnisse werden auf 0 initialisiert. Obwohl alle Kindknoten bereits generiert werden, wird nur ein zufällig ausgewählter Knoten für die Simulation verwendet. Es besteht zwar die Möglichkeit, direkt alle neuen Knoten zu simulieren, jedoch kann dies insbesondere bei Spielen mit hohem Verzweigungsgrad zu einem erheblichen Berechnungsaufwand führen. Da der Algorithmus möglicherweise diesen Teilzweig des Baumes aufgrund schlechter Simulationsergebnisse zu einem späteren Zeitpunkt ohnehin ignoriert, ist es effizienter, die Knoten nach und nach zu simulieren.

3. **Simulation** (Zeile 11): Bei der Simulation wird ausgehend von der Spielsituation im Knoten das Spiel bis zu einer Terminalstellung gespielt. Die Auswahl der Spielzüge in jeder Runde erfolgt im einfachen MCTS-Algorithmus zufällig. Am Ende wird das Spiel als gewonnen (Wert 1) oder verloren (Wert 0) bewertet.

4. **Rückpropagierung** (Zeile 12): Bei der Rückpropagierung werden alle Knoten, die auf dem Pfad vom aktuellen Knoten bis zur Wurzel liegen, aktualisiert. Dabei werden die Gewinnwerte um das Ergebnis der aktuellen Simulation erhöht und die Anzahl der Besuche in jedem Knoten um eins inkrementiert.

Die Auswahl des besten Zuges am Ende der Suchzeit (Zeile 14) kann nach verschiedenen Kriterien erfolgen:

- *Robust child*: Der Kindknoten mit den meisten Simulationen wird ausgewählt.
- *Max child*: Der Kindknoten mit dem höchsten Gewinnwert wird ausgewählt.
- *Max-robust child*: Der Kindknoten mit den meisten Simulationen und gleichzeitig dem höchsten Gewinnwert wird ausgewählt. Wenn dies nicht erfüllt werden kann, wird die Suche fortgesetzt.
- *Secure child*: Der Kindknoten, der eine gewisse untere Konfidenzgrenze maximiert, wird ausgewählt.

In dieser Arbeit werden die beiden ersten Kriterien (*robust child* und *max child*) umgesetzt und verglichen.

Abschließend lassen sich einige besondere Eigenschaften und Vorteile des MCTS-Algorithmus herausstellen.
Zunächst erfordert die einfache Umsetzung des MCTS-Algorithmus kein hohes Domänenwissen. Es müssen lediglich die Spielregeln, insbesondere die zulässigen Spielzüge und die Bedingungen für das

Spielende, bekannt sein. Im Gegensatz zum Alpha-Beta-Algorithmus sind also kein hohes strategisches Wissen und keine Erfahrung erforderlich, um eine gute Bewertungsfunktion und Zugsortierung zu implementieren. Dies ist besonders vorteilhaft in Spielen mit hoher taktischer und strategischer Komplexität, bei denen nicht alle Aspekte in der Bewertungsfunktion oder Zugsortierung abgedeckt werden können.
Weiterhin besitzt der MCTS-Algorithmus die sogenannte *anytime*-Eigenschaft. Durch das Propagieren der Ergebnisse am Ende jeder Iteration ist es jederzeit möglich, ein Ergebnis vom Algorithmus zu erhalten. Die Alpha-Beta-Suche kann ein verlässliches Ergebnis nur bei einem vollständig untersuchten Baum zurückgeeben.
Schließlich wird während der Ausführung des Algorithmus ein asymmetrischer Suchbaum aufgebaut. Durch die Auswahl des besten Kindknotens konzentriert sich der Algorithmus immer auf vermeintlich gute und interessante Spielzüge. Diese erhalten mehr Aufmerksamkeit und werden weiter untersucht. Dadurch tendiert der Suchbaum dazu, sich dem besten Spielzug anzunähern. Dieses Verhalten vermag auch der Denkweise von menschlichen Spielern mehr zu entsprechen. Diese ignorieren bei der Zugauswahl meist die nicht zielführenden Züge und konzentrieren sich auf die potentiell erfolgreichen. Eine direkte Begrenzung in der Suchtiefe existiert dadurch ebenfalls nicht.
Mit genügend Zeit und Speicherkapazität konvergiert der MCTS-Algorithmus zu dem gleichen Ergebnis (Suchbaum) wie der Alpha-Beta-Algorithmus. [6]

3.2.2. Upper Confidence Bounds for Trees (UCT)

Einer der wichtigsten Punkte des MCTS-Algorithmus ist die Auswahl des nächsten Kindknotens beim Durchlaufen des Suchbaums. Wie bereits im Kapitel 3.2.1 erwähnt, hängt die Auswahl von der *Tree Policy* ab. Diese ist entscheidend für den Aufbau des Baums und somit den Erfolg des Algorithmus. Hierbei werden in der Regel zwei Aspekte beleuchtet.

- **Exploitation:** Mit dem Begriff *Exploitation* (im Deutschen *Ausnutzung*) wird die vertiefende Untersuchung vielversprechender Züge bezeichnet. Wenn frühere Simulationen an einem Knoten bereits positive Ergebnisse geliefert haben, besteht ein großes Interesse daran, diese Ergebnisse durch weitere Simulationen zu bestätigen und den Teilbaum weiter wachsen zu lassen.
- **Exploration:** Der Aspekt der *Exploration* (im Deutschen *Erkundung*) konzentriert sich darauf, Spielzüge zu untersuchen, die bisher weniger Beachtung gefunden haben. So wird verhindert, dass Knoten mit anfangs schlechteren Simulationsergebnissen fälschlicherweise ignoriert werden, obwohl sie sich bei weiteren Simulationen positiv entwickeln könnten. Sollten sich andererseits zuvor aussichtsreiche Züge als wenig erfolgreich herausstellen, ist es von Vorteil, bereits andere Züge genauer untersucht zu haben.

Es ist wichtig, diese beiden Aspekte in eine gute Balance zubringen. [6] Hierfür wird die UCT-Formel verwendet, die wie folgt lautet:[9]

$$UCT(v) = \bar{X}_j + 2C_p\sqrt{\frac{\ln n}{n_j}} \tag{3.1}$$

wobei:

- $\bar{X}_j$ der durchschnittliche Gewinnwert ist, der von Knoten v gesammelt wurde. Der Wert befindet sich im Interval $[0, 1]$.

- C_p eine Konstante ist, die die Exploration gewichtet; es gilt $C_p > 0$.
- n_j die Anzahl der Besuche des Knotens v ist.
- n die Anzahl der Besuche des Elternknotens von v ist.

Der erste Teil der Gleichung ($\bar{X}_j$) stellt den *Exploitation*-Aspekt dar. Wenn gute Simulationsergebnisse für einen Knoten erzielt werden, wächst dieser Term.
Der zweite Teil der Gleichung $\left(2C_p\sqrt{\frac{\ln n}{n_j}}\right)$ repräsentiert dagegen den *Exploration*-Aspekt. Wenn ein Knoten häufiger simuliert wird, steigt n_j an, und somit schrumpft der Term insgesamt. Wenn hingegen andere Kindknoten des Elternknotens untersucht werden, steigt n und somit der gesamte Term. Dadurch wird der Knoten wieder interessanter für die Suche.
Wie bereits bei der Expandierung beschrieben, wird die Anzahl der Simulationen von neu generierten Knoten mit null initialisiert. In diesem Fall wird der UCT-Wert aufgrund des hinteren Terms als ∞ gewertet und automatisch als der beste Knoten angesehen. [6]
In einem Zwei-Personen-Spiel muss außerdem darauf geachtet werden, dass die Gewinnrate alterniert. Wie in Kapitel 4 beschrieben, wird das Ergebnis der Simulation bei allen Knoten aktualisiert. Das Ergebnis der Simulation wird immer aus der Perspektive des Spielers berechnet, der die Suche gestartet hat, also dem Spieler der Wurzel des Baumes. Daher stehen in allen Knoten auch die Gewinnwerte aus der Perspektive dieses Spielers. Auf Suchebenen, auf denen der gegnerische Spieler an der Reihe ist, muss daher der komplementäre durchschnittliche Gewinnwert $\left(\bar{X}_j = 1 - \frac{Gewinnwert}{n_j}\right)$ verwendet werden.
Es wird der Kindknoten ausgewählt, der den UCT- Wert maximiert.

Für eine gute Balance zwischen Exploitation und Exploration gibt es die Konstante C_p. In Spielen mit einem durchschnittlichen Gewinnwert im Intervall von $[0, 1]$ wird oft $C_p = \frac{1}{\sqrt{2}}$ verwendet. Grundsätzlich muss jedoch der optimale Wert experimentell ermittelt werden.[9]

Das UCT-Verfahren zeigt einen weiteren grundlegenden Unterschied der beiden Algorithmen auf: Im Alpha-Beta Ansatz werden Knoten endgültig abgeschnitten und nicht mehr untersucht. Im MCTS-Algorithmus werden Spielzüge nicht endgültig ignoriert. Jederzeit kann ein Zug wieder interessant und deswegen weiter untersucht werden. Dennoch ermöglicht es das UCT-Verfahren, den Fokus auf relevante Spielzüge zu legen und einen „quasi“ *Cutoff* von anderen Spielzügen zu erzeugen.[7]

3.2.3. Techniken der Simulation

Die Simulation im MCTS-Algorithmus ist ein zentrales Element. Einerseits wird angestrebt, viele Simulationen durchzuführen, um eine gute Approximation zu erzielen. Andererseits ist es ebenso wichtig, dass die Simulation aussagekräftig und realistisch ist. Im Folgenden werden zwei Techniken für die Weiterentwicklung der Simulation vorgestellt.

3.2.3.1. Zuglimitierung in der Simulation

Ein großes Problem kann sein, dass Simulationen mit zufällig ausgewählten Spielzügen deutlich mehr Halbzüge bis zu einer Terminalstellung benötigen, als es bei bewusst gewählten Spielzügen der Fall

ist. Dadurch können insgesamt weniger Simulationen durchgeführt werden. Dieses Problem tritt besonders stark auf, wenn ein Spiel einen hohen Verzweigungsgrad (so Abalone) und viele schwache Züge aufweist. Letzteres ist auch für Abalone zu bejahen, weil die meisten möglichen Spielzüge in einer Situation darin bestehen, nur eine Kugel zu bewegen und es zudem viele „Geschwisterzüge“ gibt. Das sind solche, bei denen eine oder zwei Kugeln bewegt werden, es aber einen weiteren, meist besseren Zug gibt, bei dem noch eine dritte Kugel bewegt wird.

Um dieses Problem zu umgehen, wird eine Simulationslimitierung eingeführt. Das bedeutet, dass die Simulation nach einer bestimmten Anzahl von ausgeführten Zügen abgebrochen und die Spielsituation ausgewertet wird. Dabei gibt es zwei Möglichkeiten:

Zum einen kann neben Sieg (Wert 1) oder Niederlage (Wert 0) auch die diskrete Bewertung „Unentschieden“ (Wert 0.5) eingeführt werden. Bei dieser Variante werden jedoch Spiele, bei denen ein Spieler beispielsweise fünf Kugeln vorne liegt, aber noch nicht gewonnen hat, als neutral bewertet, obwohl sich die Spielsituation für den einen Spieler deutlich besser als für den anderen darstellt.

Um dies zu vermeiden, gibt es zum anderen die Möglichkeit, eine Nicht-Terminalstellung mithilfe der Bewertungsfunktion (siehe Kapitel 3.1.2) beurteilen zu lassen. Dies hat zwar den Nachteil, dass dafür Domänenwissen benötigt wird und der Rechenaufwand höher ist, es ermöglicht jedoch eine differenziertere Betrachtung der Simulation.[16]
Der berechnete Wert der Bewertungsfunktion wird annähernd auf den Wertebereich $[0, 1]$ projiziert, indem er durch den Wert von sechs Kugeln (6000) geteilt wird. Dadurch liegt der Wert nun in der Regel im Intervall $[-1, 1]$ (gegebenenfalls wird der Wert auf die äußere Intervallgrenzen gesetzt). Anschließend wird der Wert noch auf das Intervall $[0, 1]$ angepasst. Eine genauere Projektion ist schwer möglich, da die Grenzen des Wertebereichs der Bewertungsfunktion sehr groß und kaum exakt zu bestimmen sind. Wenn eine Spielsituation als neutral gewertet wird, wird der Simulation somit der Wert 0.5 gegeben. Daraus ergibt sich, dass ein zurückgegebener Wert größer als 0.5 ist, wenn die Spielsituation positiv bewertet wird und kleiner als 0.5, wenn eine negative Bewertung vorliegt. Eine schwächere Spielsituation befindet sich also im Wertebereich $[0, 0.5[$ und eine stärkere im Bereich $]0.5, 1]$
Eine geeignete Begrenzung für die Anzahl der Halbzüge muss anhand von Experimenten ermittelt werden. Dabei müssen Aspekte wie die Anzahl der durchgeführten Simulationen und die erforderliche Aussagekraft der Simulation berücksichtigt werden.

3.2.3.2. Simulation Policy

Als weitere Technik wird eine *Simulation Policy* eingeführt. Diese beeinflusst die Auswahl der Züge bei der Simulation, welche damit nicht mehr zufällig erfolgt.[6]. Für die vorliegende Arbeit wurde der folgende Mechanismus entwickelt, um die Zugauswahl zu beeinflussen:

1. Zugsortierung: Zunächst werden alle zulässigen Züge mithilfe der Basissortierung sortiert (siehe Kapitel 3.1.5).

2. Auswahl der besten Zügen: Es wird jeweils die absolute Anzahl an besten und zufälligen Zügen aus dem angegebenen Verhältnis ermittelt. Danach wird die Anzahl an besten Zügen den späteren wählbaren Zügen hinzugefügt.

3. Auswahl von zufälligen Zügen: Anschließend wird aus den verbleibenden Zügen die Anzahl der zufälligen Züge willkürlich ausgewählt.

4. Auswählbare Züge: Die Züge aus 2 und 3 ergeben zusammen die verwendbaren Züge in der Simulation. Die Gesamtanzahl der wählbaren Züge ist dabei geringer als die Anzahl aller zulässigen Züge. Der letztendlich ausgeführte Zug wird aus den wählbaren Zügen zufällig ausgewählt.

Das genaue Verhältnis der besten und zufälligen Züge muss experimentell bestimmt werden. Zum Beispiel könnte man die besten 30 Prozent der Züge auswählen und zusätzlich 20 Prozent zufällige Züge einbeziehen. Insgesamt stehen dann 50 Prozent der Züge zur Verfügung. Mit diesem Ansatz soll vermieden werden, dass viele schwache Züge zur Auswahl stehen. Somit wird sich ein zielstrebigeres und realistischeres Spiel in der Simulation erhofft, bei dem es schneller zu einer Terminalstellung kommen und somit die Simulation vorzeitig beendet werden kann. Insgesamt ermöglicht die Kombination aus den besten und zufälligen Zügen weiterhin eine ausgewogene Berücksichtigung der Aspekte der *Exploitation* und *Exploration*.

3.2.4. Techniken des Suchbaums

Neben den Erweiterungen bei der Simulation kann innerhalb des MCTS-Algorithmus zudem versucht werden, den Aufbau des Suchbaumes zu optimieren.

3.2.4.1. Reduzierung des Verzweigungsgrades

Der erste Ansatz besteht darin, den Verzweigungsgrad zu verringern. Hierfür wird im Expandierungsschritt nur ein gewisser Anteil an Kindknoten erzeugt.[6] Dabei wird wieder angenommen, dass es bei Abalone viele schwache und ähnliche Züge gibt. Die Auswahl der Züge erfolgt nach dem in Kapitel 3.2.3 vorgestellten Mechanismus. Dadurch werden auch hier wieder die Aspekte der *Exploitation* und *Exploration* berücksichtigt. Somit kann insgesamt eine Vorauswahl der Züge getroffen werden, wodurch der Fokus mehr auf die wenigen guten Spielzüge gelegt werden kann. Ein gutes Verhältnis muss ebenfalls experimentell ermittelt werden.

3.2.4.2. Wiederverwendung des Suchbaumes

Eine weitere Methode besteht darin, einen Teil des Suchbaums aus vorherigen Suchdurchläufen wiederzuverwenden. Hierfür wird nach jeder Suche der aufgebaute Suchbaum gespeichert. Anhand des gespeicherten Spielverlaufes wird dann nachvollzogen, welcher Pfad im Baum genommen wurde. Der Teilbaum am Ende dieses Pfades kann dann für die neue Suche verwendet werden und enthält bereits die Informationen, die aus vorherigen Suchdurchläufen gesammelt wurden. Es ist jedoch zu beachten, dass möglicherweise kein Pfad gefunden wird, wenn diese Technik in Verbindung mit der Reduzierung des Verzweigungsgrades verwendet wird (siehe Kapitel 3.2.4.1). Dies kann dann daran liegen, dass die ausgeführten Züge in der letzten Suche nicht expandiert wurden.
Insgesamt verspricht diese Technik nur dann einen Vorteil, wenn eine Suche zuvor eine deutlich höhere Suchtiefe als zwei erreicht hat. Wenn die Wurzel auf Ebene null ist und der Gegner zwischen den eigenen Zugsuchen jeweils einen Zug ausführt, wird der gesuchte Teilbaum auf Ebene Zwei beginnen. Um nun einen Vorteil zu haben, sollte dieser Teilbaum bereits eine gewisse Größe haben. [2]

4. Durchführung der Tests

Im Folgenden wird die technische Testumgebung sowie der generelle Ablauf der Tests beleuchtet.

Die Tests werden auf dem Computercluster Hydra der Technischen Universität Berlin durchgeführt. Die genaue Konfiguration kann der Abbildung A.1 im Anhang entnommen werden. Dabei wird die Programmiersprache Python (Version 3.10.12) verwendet.

Für die Bewertung der einzelnen Techniken im Alpha-Beta-Algorithmus wird als Metrik die Anzahl der untersuchten Zustände und die erreichte Suchtiefe verwendet. Insbesondere bei diesem deterministischen Algorithmus ist es wichtig, die Techniken in verschiedenen Spielstellungen zu testen. Hierfür werden die beiden vorgestellten Startstellungen aus Kapitel 2.1 sowie zwei Mittelspielstellungen benutzt.
Bei Techniken wie dem *Aspiration Window*, der Ruhesuche und der Bewertungsfunktionen, bei denen die Anzahl der untersuchten Knoten keine oder nur teilweise Rückschlüsse auf die Spielstärke gibt, werden Spiele simuliert und die Gewinnhäufigkeit ausgewertet. Auch hier ist es wichtig, verschiedene Spielsituationen für die Simulation zu verwenden. Hierfür werden die Tests auf den beiden Startaufstellungen und 18 weiteren neutralen Spielsituationen, die sich noch im Anfangsstadium des Spiels befanden, durchgeführt.

Eine Metrik für die Beurteilung der einzelnen Techniken im MCTS-Algorithmus zu finden, stellt sich als schwieriger dar. Dies liegt vor allem daran, dass viele Techniken zunächst einen zusätzlichen Rechenaufwand mit sich bringen, aber dennoch zu einem stärkerem Spielverhalten führen können.
Zudem kann eine höhere Anzahl von Simulationen auf ein besseres Ergebnis hindeuten, aber muss dies nicht, wenn dadurch zum Beispiel die Qualität der Simulation beeinträchtigt wird.[6]. Die Anzahl der Simulationen wird nur bei der Zuglimitierung in der Simulation als Metrik verwendet. Alle weiteren Techniken werden in Form von Spielen ausgewertet.

Insgesamt werden bei beiden Algorithmen die einzelnen Techniken bzw. Parameter nacheinander systematisch ausgewertet und gegebenenfalls in den Algorithmus integriert. Hierbei wird davon ausgegangen, dass die Parameter und Techniken als unabhängig voneinander betrachtet werden können. Für den Vergleich der beiden Algorithmen werden die beiden zuvor vielversprechendsten ermittelten Konfigurationen gegeneinander in Spielen getestet.

Grundsätzlich wird in Tests, in denen nur eine Suche auf einer Spielstellung durchgeführt wurde, eine Suchzeit von 1.000 Sekunden zur Verfügung gestellt, um die Technik möglichst tiefgehend zu testen. In Tests, in denen die Ergebnisse von Spielen als Metrik dienen, hat jeder Spieler 20 Sekunden pro Halbzug Zeit. Ein Spiel wird zudem nach 150 Halbzügen beendet. Wenn bis dahin eine Farbe führt, wird dieses Spiel als Sieg für diese und ansonsten als Unentschieden gewertet. Es werden immer 40 Spiele simuliert.

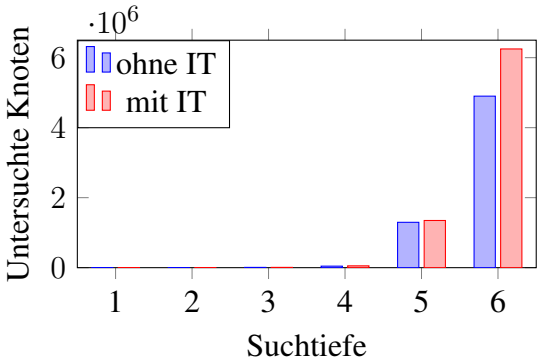
5. Ergebnisse

Im Nachfolgenden werden die Ergebnisse der einzelnen Tests dargestellt. Zunächst werden die beiden Algorithmen für sich betrachtet. Abschließend werden die Ergebnisse des direkten Vergleichs vorgestellt.

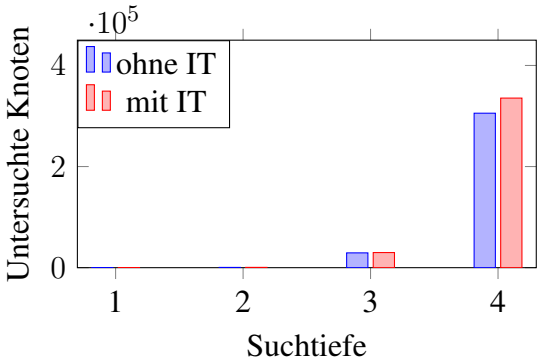
5.1. Alpha-Beta-Suche

In diesem Abschnitt werden die Ergebnisse der verschiedenen Techniken des Alpha-Beta-Algorithmus behandelt. Für die Übersichtlichkeit werden immer nur die Ergebnisse von zwei Spielstellungen präsentiert. Die Ergebnisse der anderen Spielstellungen können im Anhang betrachtet werden.

In den Abbildungen 5.1 und A.2 werden die Auswirkungen der iterativen Tiefensuche auf den Alpha-Beta-Algorithmus dargestellt. Das Balkendiagramm zeigt die Anzahl der untersuchten Knoten pro Suche für eine bestimmte Suchtiefe. Es ist deutlich erkennbar, dass die iterative Tiefensuche zwar eine höhere Anzahl von Knoten untersucht, jedoch der zusätzliche Aufwand vergleichsweise gering ist und sich hauptsächlich bei höheren Tiefen bemerkbar macht.
Die nachfolgenden Darstellungen konzentrieren sich jeweils nur auf die letzten relevanten Tiefen.



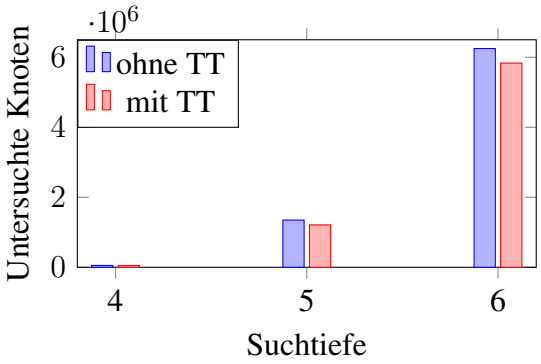
(a) Standardaufstellung



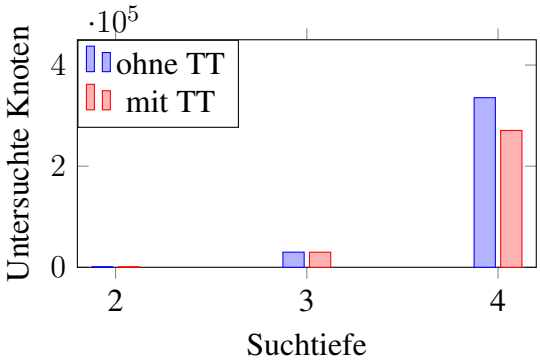
(b) Mittelspielstellung 1

Abbildung 5.1.: Vergleich 1 mit und ohne iterative Tiefensuche (IT)

Die Testergebnisse der *Transposition Table* werden in den Abbildungen 5.2 und A.3 dargestellt. Insgesamt ist hier eine Reduktion der untersuchten Knoten zu erkennen. Bei der Untersuchung der Belgium-Daisy-Stellung (siehe Abbildung 2.1b) kann allerdings auch ein minimaler Mehraufwand bei einer Suchtiefe von 5 festgestellt werden. Dennoch wird die *Transposition Table* in den weiteren Experimenten verwendet. Mit einer Kollisionsrate, die stets unter 0,05 Prozent liegt, scheint die angenommene Größe der Tabelle ausreichend zu sein.



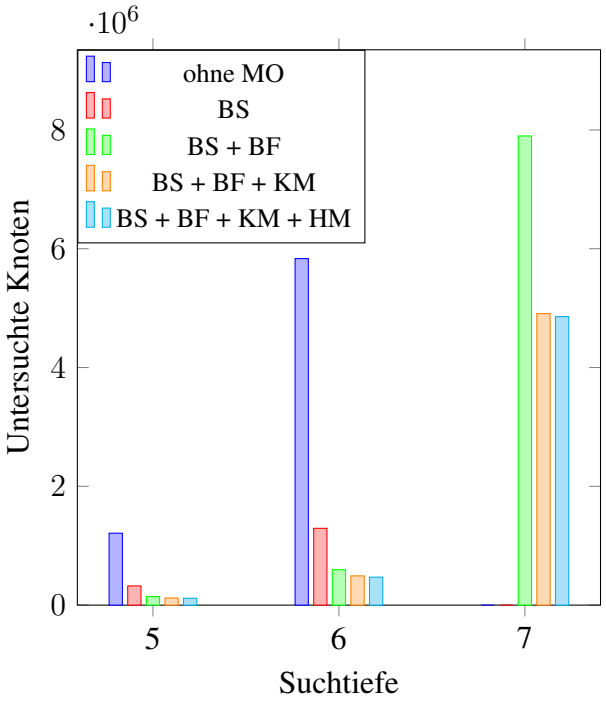
(a) Standardaufstellung



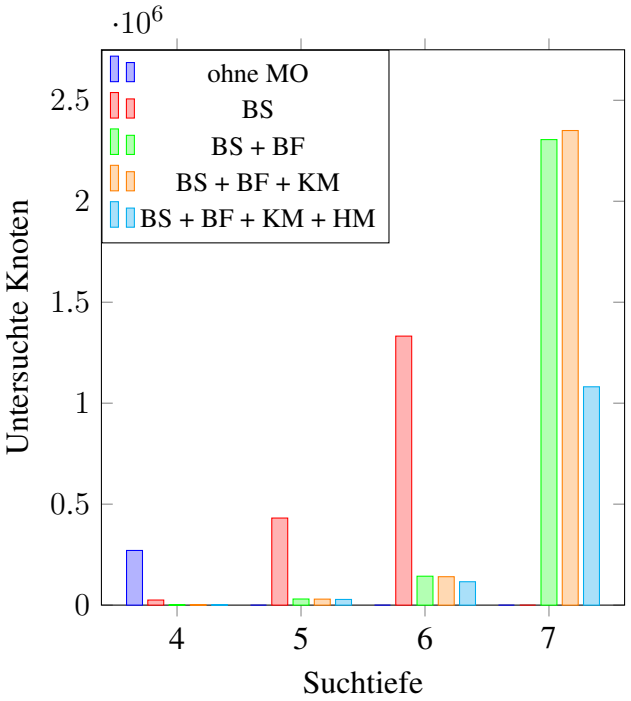
(b) Mittelspielstellung 1

Abbildung 5.2.: Vergleich 1 mit und ohne *Transposition Table* (TT)

In den Abbildungen 5.3 und A.4 werden die Ergebnisse der Zugsortierung dargestellt. Zunächst wird die Basissortierung (BS), gefolgt von der Sortierung mit der Bewertungsfunktion (BF), hiernach die *Killer-Move*-Sortierung (KM) und abschließend die Anwendung des *Hash-Moves* (HM) untersucht. Es ist zu beachten, dass die Techniken teilweise unterschiedliche Tiefen erreichen. Techniken, die eine bestimmte Tiefe nicht innerhalb der vorgegebenen Zeit erreichen, werden in dieser Tiefe mit null bewertet.



(a) Standardaufstellung



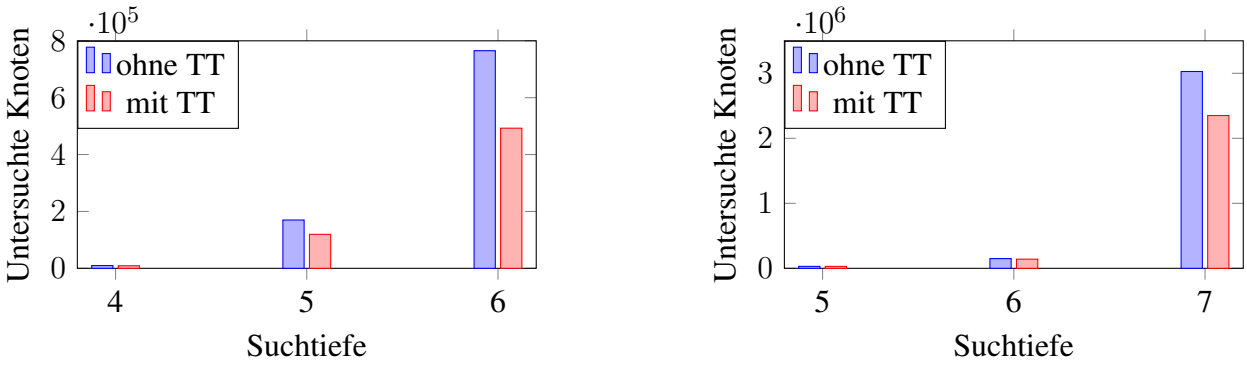
(b) Mittelspielstellung 1

Abbildung 5.3.: Vergleich 1 der Zugsortierung (move ordering, MO)

Grundsätzlich ist zu erkennen, dass die Basissortierung bereits eine Reduzierung der untersuchten Zustände bewirkt. Die weiteren Techniken reduzieren teilweise den Aufwand noch einmal stark, wo-

bei die Killer-Move-Sortierung und der Hash-Move auch kleine Verschlechterungen mit sich bringen können (siehe zum Beispiel Suchtiefe 7 in Abb. 5.3b und Suchtiefe 8 in Abb. A.4a). Diese sind jedoch gering und oft auch nur in großen Tiefen festzustellen, weshalb im Folgenden mit der Zugsortierung BS + BF + KM + HM weitergearbeitet wird.

Die Abbildung 5.4 und A.5 zeigen erneut die Auswirkung durch die Verwendung der *Transposition Table*. Dieses Mal wird jedoch die Zugsortierung BS + BF + KM angewandt. Zwecks der Vergleichbarkeit unterbleibt die Anwendung des Hash-Moves. Im Zusammenspiel mit dieser Zugsortierung ist eine deutlichere Reduzierung der untersuchten Knoten durch die *Transposition Table* erkennbar. Sogar bei der Belgium-Daisy-Stellung, bei der es ohne Zugsortierung zu einem Mehraufwand kommt, ist ein positiver Effekt der *Transposition Table* nun zu bemerken (siehe Abb. A.5a).



(a) Standardaufstellung

(b) Mittelspielstellung 1

Abbildung 5.4.: Vergleich 1 mit und ohne *Transposition Table* (TT) und Zugsortierung (BS + BF + KM)

Die Ermittlung einer geeigneten Fenstergröße bzw. eines Offsets für die Technik des *Aspiration Window* erweist sich als kompliziert. Zunächst wird für eine grobe Einschätzung der Größenordnung der Verlauf des Spielwertes während des Spiels betrachtet. Die Abbildung 5.5 zeigt einen Ausschnitt hiervon, wobei der Spielwert aus der Perspektive einer Farbe über den Verlauf der Spielzüge aufgetragen ist. Es ist zu erkennen, dass der Wert meistens leicht alternierend verläuft, wobei dessen Veränderung außer beim Schlagen einer Kugel gering ist. Die Fenstergrößen 40, 70, 100, 200 und 400 werden weiter untersucht, da sie sich in einer vielversprechenden Größenordnung befinden.

Die Fenstergrößen wird zunächst auf den vier einzelnen Spielstellungen getestet, um den Effekt der Reduzierung der untersuchten Knoten nachzuweisen. Die Abbildungen 5.6 und A.6 zeigen die Ergebnisse für die verschiedenen Fenstergrößen. Die Ergebnisse weisen keine klare Tendenz für die Parameterwahl auf. Insgesamt kann aber für die Fenstergrößen 40, 70 und 100 in den meisten Fällen eine Reduktion des Suchaufwandes festgestellt werden. Die Fenstergrößen 200 und 400 weisen dagegen teilweise eine Erhöhung der untersuchten Knoten auf (siehe Abb. 5.6a und A.6a). Aber auch durch die Fenstergrößen 40 und 70 entsteht in der Belgium-Daisy-Stellung (A.6a) ein Mehraufwand .

Für eine aussagekräftigere Auswertung der Technik wird diese zudem in Spielen getestet. Dabei spielt ein Spieler (P1) immer ohne die Verwendung des *Aspiration Window*, während der andere Spieler (P2)

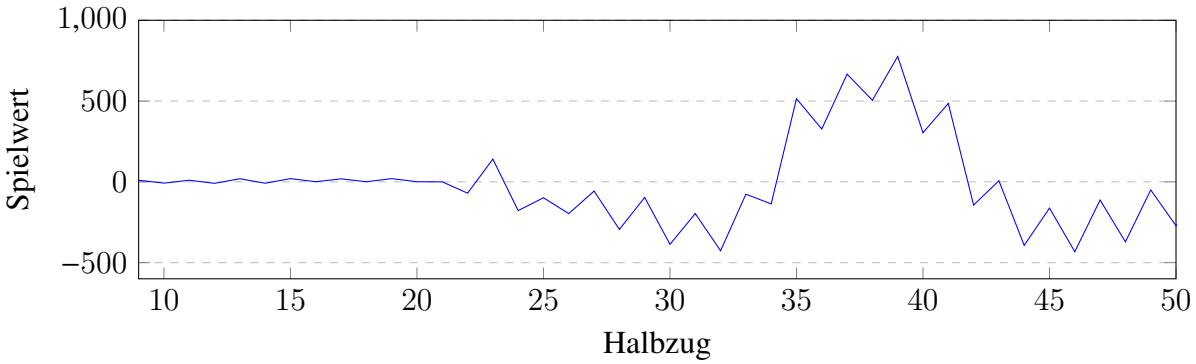
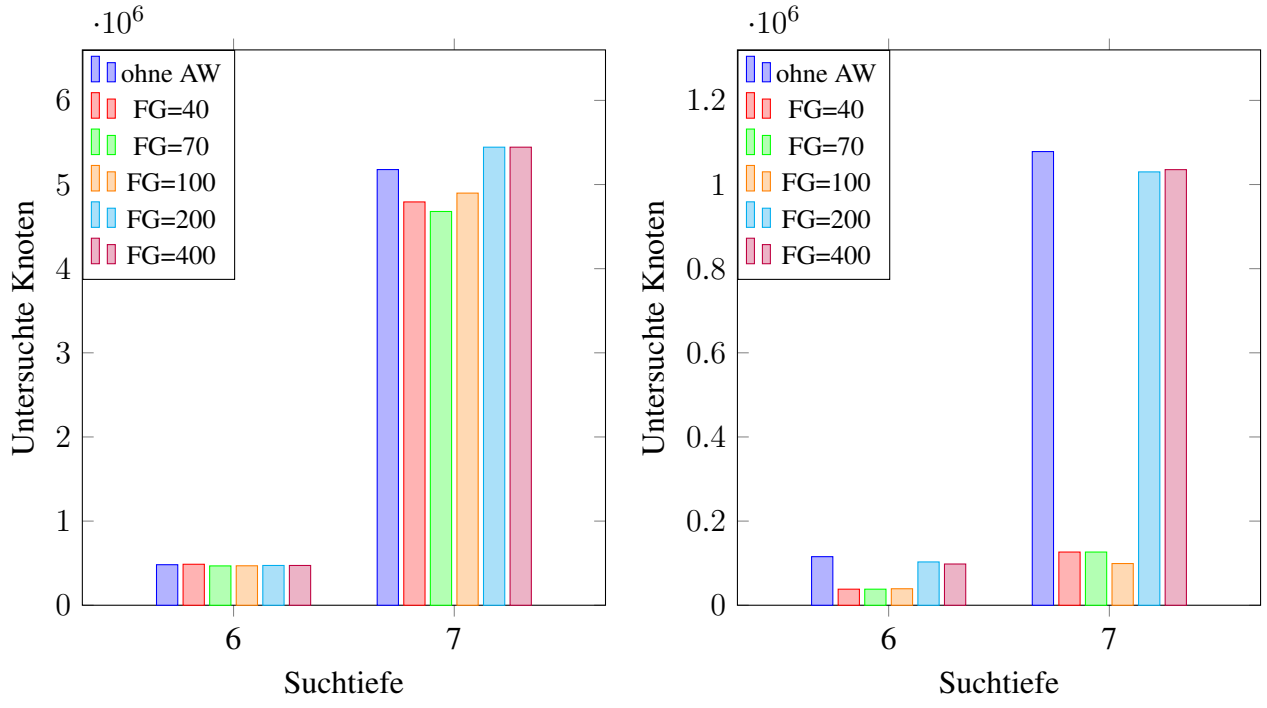


Abbildung 5.5.: Ausschnitt des Verlaufes des Spielwertes aus Sicht einer Farbe innerhalb eines Spiels



(a) Standardaufstellung

(b) Mittelspielstellung 1

Abbildung 5.6.: Vergleich 1 mit und ohne *Aspiration Window* (AW) für verschiedene Fenstergrößen (FG)

das *Aspiration Window* mit den verschiedenen Fenstergrößen einsetzt.
Die Tabelle 5.1 zeigt die Ergebnisse der Spiele. In der Tabelle wird die relative Sieg- (S) bzw. Unentschiedenhäufigkeit (U) dargestellt. Durch die Möglichkeit eines Unentschiedens, kann es im Folgenden auch vorkommen, dass eine Sieghäufigkeit von unter 50 Prozent als vielversprechend angesehen wird. Weiterhin wird zur differenzierten Betrachtung auch die durchschnittliche Anzahl der Punkte der Spieler pro Spiel ermittelt, wobei ein Punkt einer geschlagenen Kugel des Gegners entspricht.
Es wird deutlich, dass das Spielen mit *Aspiration Window* und einer Fenstergröße von 100 mit 60 Prozent gewonnener Spiele am vielversprechendsten ist. Auch die Fenstergrößen 40 und 200 können eine positive Gewinnrate (52,5 Prozent und 57,5 Prozent) aufweisen, wobei die Fenstergröße von 100 mit 4,6 zu 3,3 Punkten pro Spiel auch hier die besten Werte aufweist. Im Gegensatz dazu sind die Fenstergrößen 70 und 400 gegen einen Algorithmus, der das *Aspiration Window* nicht anwendet, im Nachteil. Die folgenden Tests werden mit dem *Aspiration Window* und einer Fenstergröße von 100 durchgeführt.

	Spielkonfiguration				
	FG=40	FG=70	FG=100	FG=200	FG=400
S P1/U/S P2 in %	35/12,5/52,5	50/12,5/37,5	30/10/60	35/7,5/57,5	47,5/15/37,5
Punkte P1 / Punkte P2	3,3 / 4,1	3,5 / 3,8	3,3 / 4,6	4,0 /4,4	4,3 / 4,0

Tabelle 5.1.: Vergleich der Spielergebnisse mit und ohne *Aspiration Window* für verschiedene Fenstergrößen (FG)

Die Veränderung der Spielstärke durch die Verwendung der Ruhesuche wird ebenfalls mithilfe von Spielen getestet. Der Spieler P1 spielt dabei ohne und der Spieler P2 mit der Ruhesuche. Die Tabelle 5.2 stellt die Ergebnisse dar. Der Spieler ohne Ruhesuche (P1) hat dabei mit 45 Prozent einen leicht höheren Anteil an gewonnenen Spielen. Mit Verwendung der Ruhesuche werden dagegen 40 Prozent der Spiele für sich entschieden. Bei der Betrachtung der durchschnittlich geschlagenen Kugeln ist mit 4,0 zu 3,9 der Unterschied ebenfalls gering. Eine klare Tendenz ist nicht zuerkennen. Da die Ruhesuche aber in den Ergebnissen keinen klaren Vorteil bringt, wird sie nicht weiter verwendet.

	Spielkonfiguration
	ohne RS vs. mit RS
S P1/U/S P2 in %	45/15/40
Punkte P1 / Punkte P2	4,0 / 3,9

Tabelle 5.2.: Vergleich der Spielergebnisse mit und ohne Ruhesuche (RS)

Zuletzt wird die Bewertungsfunktion getestet. Hierfür spielt jeweils ein Spieler (P1) mit der neutralen Gewichtung der Bewertungsfunktion, bei der alle Zusatzgewichte (vgl. Kapitel 3.1.2) den Wert 1

haben. Der andere Spieler (P2) spielt dagegen mit Zusatzgewichten, die zu einer offensiveren bzw. defensiveren Spielweise führen. In beiden Fällen kann sich aber die neutrale Spielweise mit 50 Prozent bzw. 65 Prozent an gewonnen Spielen durchsetzen (siehe Tabelle 5.3). Auch die durchschnittlich erzielten Punkte (4,3 zu 3,3 und 4,8 zu 3,9) sprechen für die neutrale Spielweise.
Anschließend wird überprüft, ob und inwieweit sich die Veränderungen der Bewertungsfunktion gegenüber der ursprünglichen Vegleichsfunktion auswirken. Die Ergebnisse in der Tabelle 5.3 zeigen eine leicht verbesserte Spielstärke (45 Prozent zu 37,5 Prozent gewonnener Spiele) bei Benutzung der in dieser Arbeit entwickelten Bewertungsfunktion.
Daher wird in dem abschließenden Vergleich mit dem MCTS-Alogrithmus die Bewertungsfunktion dieser Arbeit mit neutralen Zusatzgewichten verwendet.

	Spielkonfiguration		
	Normal vs. Offensiv	Normal vs. Defensiv	Normal vs. Vergleichsfkt.
S P1/U/S P2 in %	50/17,5/32,5	65/7,5/27,5	45/17,5/37,5
Punkte P1 / Punkte P2	4,3 / 3,3	4,8 / 3,9	4,2 / 4,0

Tabelle 5.3.: Vergleich der Spielergebnisse der verschiedenen Bewertungsfunktionstypen

5.2. Monte-Carlo Tree Search

Im Folgenden werden die Ergebnisse des MCTS-Alogrithmus präsentiert. Der Basisalgorithmus startet ohne Simulationslimitierung, Simulation Policy, Reduzierung des Verzweigungsgrad oder Wiederverwendung von Suchbäumen. Für die Konstante C in der UCT-Formel wird zunächst der Wert 1,4 angenommen. Die Auswahl des besten Zuges am Ende des Algorithmus erfolgt erst einmal über den Gewinnwert. Nachfolgend werden die einzelnen Parameter ausgewertet und gegebenenfalls in den Algorithmus mit aufgenommen.

In der Tabelle 5.4 ist je Startaufstellung die Anzahl der Simulationen für die verschiedenen Zuglimitierungen pro Simulation (vgl. Kapitel 3.2.3.1) dargestellt.

Startaufstellung	**Anzahl der Simulationen**				
	keine ZL	ZL=10	ZL=40	ZL=80	ZL=150
Standard	903	109.610	28.404	13.960	7.465
Belgium-Daisy	1.045	126.885	29.709	14.683	7.958

Tabelle 5.4.: Vergleich der Zuglimiterung (ZL) in der Simulation

Normalerweise dauert ein Abalone-Spiel im Durchschnitt 87 Halbzüge (vlg. Kapitel 2.3). Dagegen dauert ein randomisiertes Spiel ohne Limitierung meist zwischen 300 und 2.800 Halbzügen. Dadurch

können lediglich um die 1.000 Simulationen ausgeführt werden. Dies entspricht einer Simulation pro Sekunde. Bei einer starken Limitierung von beispielsweise zehn Halbzügen können am meisten Simulationen (über 100.000) erreicht werden, allerdings ist hier auch die Simulation am wenigsten aussagekräftig. Wie bereits erwähnt ist in Abbildung 5.5 zu sehen, dass der Spielwert oft über mehrere Spielzüge um einen Wert alterniert. Die Tendenz für eine Verbesserung oder Verschlechterung einer Spielsituation durch einen Zug ist daher oft nicht direkt oder erst nach mehreren Spielzügen zu erkennen. Um möglichst viele aber gleichzeitig aussagekräftige Simulationen zu erhalten, wird daher in den weiteren Experimenten ein Halbzuglimit von 40 festgelegt. Dies entspricht bei einer Simulationszeit von 20 Sekunden je nach Spielsituation um die 600 Simulationen pro Suche.

Als nächstes werden in den Tabellen 5.5 und 5.6 die Ergebnisse für verschiedene Werte der Konstante C in der UCT-Formel (vgl. Formel 3.1) präsentiert. Hierbei werden zunächst die Werte 0,4 sowie 0,9 und 1,4 getestet (siehe Tabelle 5.5). Es ist zu erkennen, dass der Algorithmus für C=0,4 am stärksten spielt. Diese Konfiguration gewinnt mit 90 Prozent und 100 Prozent fast alle Spiele gegen Algorithmen mit C=0,9 bzw. C=1,4.

	Spielkonfiguration		
	C=0,4 vs. C=0,9	C=0,4 vs. C=1,4	C=0,9 vs. C=1,4
S P1/U/S P2 in %	90/10/0	100/0/0	45/55/0
Punkte P1 / Punkte P2	3,3 / 4,1	3,5 / 3,8	3,3 / 4,6

Tabelle 5.5.: Vergleich 1 der Spielergebnisse für verschiedene Konstanten C

Da die Ergebnisse sehr deutlich sind, wird noch ein weiterer Test mit den Werten 0,2 sowie 0,4 und 0,6 für die Konstante C durchgeführt, um sich dem optimalen Wert weiter zu nähern. Die Tabelle 5.6 beinhaltet die Ergebnisse. In diesem Fall spielt der Algorithmus mit C=0,2 hinsichtlich einer Siegquote von 55 Prozent und 60 Prozent am stärksten. In den folgenden Tests wird daher dieser Wert verwendet.

	Spielkonfiguration		
	C=0,4 vs. C=0,2	C=0,4 vs. C=0,6	C=0,2 vs. C=0,6
S P1/U/S P2 in %	25/20/55	40/25/35	60/15/25
Punkte P1 / Punkte P2	3,3 / 4,1	3,5 / 3,8	4,6 / 3,3

Tabelle 5.6.: Vergleich 2 der Spielergebnisse für verschiedene Konstanten C

Weitergehend wird überprüft, welches Verfahren für die Auswahl des besten Zuges am Ende des Algorithmus am stärksten ist (vgl. Kapitel 3.2.1). Hierbei wählt ein Spieler (P1) den Zug über den höchsten Gewinnwert (*max child*) und der andere (P2) über die meisten Simulationen (*robust child*).

Durch die Ergebnisse in der Tabelle 5.7 wird deutlich, dass die Wahl über *robust child* mit 65 Prozent gewonnener Spiele am vielversprechendsten ist. Im Folgenden wird dieser Auswahlprozess verwendet.

	Spielkonfiguration
	max child vs. *robust child*
S P1/U/S P2 in %	17,5/17,5/65
Punkte P1 / Punkte P2	2,0 / 3,2

Tabelle 5.7.: Vergleich der Spielergebnisse für die Auswahl der besten Aktion

Bei der Verwendung der *Simulation Policy* entsteht durch das Sortieren der Züge bei der Simulation ein Mehraufwand. Dadurch werden ungefähr vier Prozent weniger Simulationen durchgeführt. In der Tabelle 5.8 sind die Spielergebnisse aufgeführt. Der Spieler P1 spielt dabei ohne und der Spieler P2 mit der *Simulation Policy*. Es werden drei verschiedene Verhältnisse getestet. Eine Konfiguration von (0,3 / 0,2) bedeutet, dass die Auswahl der Züge aus den 30 Prozent besten und 20 Prozent zufällig gewählten Zügen besteht. Insgesamt wird die Anzahl möglicher Spielzüge somit um 50 Prozent reduziert.
Die Ergebnisse zeigen, dass der Spieler ohne *Simulation Policy* mit 55 und 47,5 Prozent mehr Spiele gewinnt. Lediglich die Konfiguration (0,4 / 0,1) spielt ebenbürtig, denn beide Spieler gewinnen 40 Prozent der Spiele. Nachfolgend wird daher keine *Simulation Policy* angewendet.

	Spielkonfiguration		
	(0,25 / 0,25)	(0,3 / 0,2)	(0,4 / 0,1)
S P1/U/S P2 in %	47,5/22,5/30	55/12,5/32,5	40/20/40
Punkte P1 / Punkte P2	3,7 / 3,2	3,2 / 3,0	3,8 / 3,7

Tabelle 5.8.: Vergleich der Spielergebnisse mit und ohne *Simulation Policy*

In der Tabelle 5.9 sind die Ergebnisse für die Reduzierung des Verzweigungsgrades im Suchbaum dargestellt. Für die verschiedenen Spielkonfigurationen gilt das Gleiche wie bei den Tests der *Simulation Policy*. Der Verzweigungsgrad wird insgesamt um 50 Prozent gesenkt. Der Spieler P1 spielt ohne und der Spieler P2 mit Reduzierung des Verzweigungsgrades. Der Spieler P2 spielt in allen Fällen mit 75 Prozent, 55 Prozent und 77,5 Prozent gewonnenen Spielen stärker als Spieler P1. Im Folgenden wird die Reduzierung des Verzweigungsgrades mit einer Konfiguration von (0,4 / 0,1) angewandt.

Abschließend wird die Wiederverwendung des Suchbaums aus vorherigen Suchen getestet. Die Ergebnisse sind in der Tabelle 5.10 dargestellt. Dabei spielt Spieler P1 ohne und Spieler P2 mit Wiederverwendung des Teilsuchbaums.

	Spielkonfiguration		
	(0,25 / 0,25)	(0,3 / 0,2)	(0,4 / 0,1)
S P1/U/S P2 in %	20/5/75	35/10/55	17,5/10/77,5
Punkte P1 / Punkte P2	3,0 / 4,3	3,6 / 4,3	3,2 / 4,5

Tabelle 5.9.: Vergleich der Spielergebnisse mit und ohne Reduzierung des Verzweigungsgrades

Durch die Wiederverwendung des Suchbaums kann keine klare Verbesserung der Spielstärke ermittelt werden. Zwar werden 50 Prozent der Spiele gewonnen, dennoch gewinnt auch der Spieler ohne Anwendung dieser Technik 45 Prozent der Spiele. Bei der Betrachtung der durchschnittlichen Punkte ist mit 4,0 zu 4,5 Punkten eine etwas deutlichere Tendenz für die Wiederverwendung des Suchbaums zu erkennen. In den meisten Suchen wird eine Baumtiefe von maximal drei erreicht.
Da insgesamt klare Verschlechterung der Spielstärke zu erkennen ist, wird die Technik in den nächsten Tests weiter benutzt.

	Spielkonfiguration
	P1 vs. P2
S P1/U/S P2 in %	45/5/50
Punkte P1 / Punkte P2	4,0 / 4,5

Tabelle 5.10.: Vergleich der Spielergebnisse mit und ohne Wiederverwendung des Teilsuchbaumes

5.3. Vergleich von Alpha-Beta-Suche und Monte-Carlo Tree Search

Abschließend wird die aus den beiden vorherigen Kapiteln 5.1 und 5.2 vielversprechendsten Zusammenstellungen der Algorithmen direkt in Spielen miteinander verglichen.

Bei der Alpha-Beta-Suche wird dabei die iterative Tiefensuche, die *Transpostion Table*, die Zugsortierung BS + BF + KM + HM, das *Aspiration Window* mit einer Fenstergröße von 100 und die in dieser Arbeit entwickelte Bewertungsfunktion mit einem neutralen Spielverhalten verwendet.
Bei dem MCTS-Algorithmus wird wiederum eine Zuglimitierung von 40 Halbzügen in der Simulation, eine Reduzierung des Verzweigungsgrades (Verhältnis: (0,4 / 0,1)) und die Wiederverwendung des Suchbaums angewandt. Für die UCT-Konstante gilt der Wert 0,2 und der beste Spielzug wird über die höchste Anzahl an Simulationen ausgewählt.

Die Tabelle 5.1 stellt die Ergebnisse dar. Als Teststellungen sind die beiden bekannten Startaufstellungen gewählt worden. Spieler P1 verwendet die Alpha-Beta-Suche, während Spieler P2 die besten

Züge mithilfe des MCTS-Algorithmus bestimmt.
Der Alpha-Beta-Algorithmus gewinnt dabei 100 Prozent der Spiele. Durch die Betrachtung der durchschnittlich erzielten Punkte (6 Punkte zu 1,5 Punkten) ist zudem zu erkennen, dass die meisten Spiele mit einer deutlichen Punktedifferenz beendet werden.

	Spielkonfiguration
	Alpha-Beta-Suche vs. MCTS
S P1/U/S P2 in %	100/0/0
Punkte P1 / Punkte P2	6 / 1,5

Tabelle 5.11.: Vergleich Alpha-Beta-Suche vs. MCTS

6. Diskussion

Im Nachfolgendem werden die Ergebnisse aus Kapitel 5 eingeordnet, mit bestehender Literatur verglichen und Besonderheiten diskutiert und erklärt. Zudem werden die Limitierungen der Arbeit und die daraus folgenden Weiterentwicklungen sowie Untersuchungsbereiche beleuchtet.

6.1. Einordnung der Ergebnisse der Alpha-Beta-Suche

Die Ergebnisse des Alpha-Beta-Algorithmus stimmen größtenteils mit der bereits existierenden Literatur wie zum Beispiel den Arbeiten von Pascal Chorus [16], Michiel Verloop [19] und der Autorengruppe um Athanasios Papadopoulos[15] überein.

Auffällig sind teilweise die Ergebnisse der *Transposition Table*. Zwar kann insbesondere im Zusammenwirken mit einer guten Zugsortierung eine Reduzierung des Suchaufwandes ermöglicht werden. Allerdings kommt es ohne das Anwenden der Zugsortierung in der Startstellung Belgium-Daisy zu mehr untersuchten Knoten als beim Nicht-Verwenden der *Transposition Table* (siehe Abb. A.3a).
Ein Erklärungsansatz für dieses Verhalten basiert auf der Beobachtung, dass in dem Spiel Abalone Spielzüge, die eine Verbesserung des Spielwertes bewirken, oft mit einem ähnlich starken Spielzug des Gegners in der nächsten Aktion ausgeglichen werden. Dies geschieht insbesondere bei neutralen Spielstellungen. Dadurch entsteht ein alternierender Spielwert (siehe Abb. 5.5) bzw. je nach Spielsituation ein Spielwert, der bei einem Halbzug positiv und im nächsten Halbzug neutral ist. Wenn nun das Ergebnis in der *Transposition Table* beispielsweise auf dem letzten Halbzug der eigenen Farbe basiert, aber das ohne die *Transposition Table* ermittelte Ergebnis auf der Farbe des Gegners basieren würde, können die Alpha-Beta-Schranken unterschiedliche Werte haben. Dies kann möglicherweise zu weniger *Cutoffs* bei der Verwendung der *Transposition Table* führen. Hier bedarf es somit weiteren Untersuchungen und gegebenenfalls Anpassungen in der Implementation der *Transposition Table* oder der Bewertungsfunktion.

Im Bereich der Zugsortierung kann die Stärke der Basissortierung bestätigt werden. Ebenfalls führt die Sortierung mithilfe der Bewertungsfunktion zur Reduktion des Suchaufwandes.
Das Sortieren der Züge mithilfe von Killer-Moves und Hash-Moves ist dagegen - wie gezeigt - stark von der Spielsitiuation abhängig und führt nicht unbedingt zu einer Reduzierung des Suchaufwandes. Hier bedarf es weiterer Untersuchungen, in welchen Situationen die Techniken einen Vorteil und in welchen sie einen Nachteil bringen. Gerade in unruhigen Situationen kann das Zurückgreifen auf das Wissen von vorherigen Suchiterationen eventuell auch negative Effekte mit sich bringen. Abhilfe könnte hier eine Spielphasenerkennung schaffen, an die die Zugsortierung angepasst werden kann.

Für die Anwendung des *Aspiration Window* stellt sich als geeignete Fenstergröße 100 heraus. Den-

noch könnte auch hier eine Spielphasenerkennung zu einer effektiveren Fenstergröße führen. Die Ergebnisse der im Einzelnen untersuchten Spielstellungen (siehe Abb. 5.6 und A.6) zeigen, dass eine passende Fenstergröße durchaus von der Spielsituation abhängt und es bei ungünstig gewählten Fenstergrößen zu erheblichem Mehraufwand kommen kann (siehe Abb. A.6a), weil die Suche neu gestartet werden muss. So vermag in ruhigeren Spielphasen ein kleineres Fenster effektiver zu sein. Bei unruhigen Spielsituationen dagegen könnte die Untersuchung mit einem großen Fenster mehr Erfolg versprechen. Auch die Einbeziehung des Spielverlaufs und das Veränderungsverhalten des Spielwertes könnten eine effektivere Nutzung des *Aspiration Window* ermöglichen.

Was die Verwendung der Ruhesuche betrifft, kann - anders als in der Arbeit von Pascal Chorus [16]- keine klare Verbesserung der Spielstärke erzielt werden. Im Gegenteil: Durch den vermutlich zusätzlichen Suchaufwand spielt der Algorithmus mit Ruhesuche sogar leicht schwächer. Die Unterschiede sind aber minimal und nicht eindeutig. Insbesondere eine Verbesserung ist aber nicht zuerkennen.

Die Veränderungen der Bewertungsfunktion im Vergleich zu der Version aus der Arbeit von Pascal Chorus [16] kann wiederum eine leichte Verbesserung der Spielstärke bewirken. Der Unterschied der Gewinnrate ist allerdings gering, weshalb weitere Untersuchungen für eine abschließende Bewertung nötig sind. Hier könnten auch zusätzliche Veränderung der Bewertungskriterien zu mehr Erfolg führen. Eine Möglichkeit wäre beispielsweise, wie von Aichholzer, Aurenhammer und Werner vorgeschlagen, als Zentrum des Spiels nicht die Feldmitte sondern den Schwerpunkt der Kugeln zu nehmen. [1]
Die in dieser Arbeit benutzten Gewichtungen der einzelnen Kriterien können als geeignet angesehen werden. Sowohl offensivere als auch defensivere Spielweisen sind gegenüber der neutralen Gewichtung von Nachteil. Dennoch sei hier zu bedenken, dass eine offensivere Spielweise im Zusammenwirken mit Techniken wie der Ruhesuche wiederum von Vorteil sein könnte.

Insgesamt können die Ergebnisse aus der bestehenden Literatur größtenteils bestätigt werden. Durch die Weiterentwicklung der Bewertungsfunktion und der Zugsortierung sowie dem anschließenden systematischen Vergleich der Techniken ist somit eine spielstarke Alpha-Beta-Suche entwickelt worden.

6.2. Einordnung der Ergebnisse der MCTS

Anders als für den Alpha-Beta-Algorithmus gibt es für den MCTS-Algorithmus im Zusammenhang mit dem Spiel Abalone keine weitreichende Literatur. Lediglich die Umsetzung einer Monte-Carlo-Suche wurde in der Arbeit von Pascal Chorus [16] bereits untersucht.

Wie in jener Arbeit ist auch bei der Umsetzung des MCTS-Algorithmus eine Limitierung der Anzahl an ausgeführten Zügen in der Simulation nötig, da ansonsten durch die lange Laufzeit zu wenig Simulationen ausgeführt werden können. Auf Basis des Spielwertverlaufs und der Anzahl an erreichten Simulationen wird eine Zuglimitierung von 40 Halbzügen in der Simulation als geeignet angesehen. Hierbei ist zu bedenken, dass es sich nur um Annahmen handelt und keine weiteren Test in Form von Spielen stattgefunden haben. Ebenfalls muss die Technik noch einmal genauer im Zusammenhang einer *Simulation Policy*, die zu einen anderem Simulationsverhalten führt, überprüft werden.

Als interessante Erkenntnis stellt sich auch der geringe Wert für die UCT-Konstante C (in der Literatur wird oft der Wert 1.4 angenommen) und die klare Überlegenheit des Auswahlverfahrens für die beste Aktion anhand der Anzahl der Simulationen heraus. Beides deutet darauf hin, dass Untersuchungen über das Aufbauverhalten des Suchbaumes weitere Erkenntnisse für die Verbesserung des Algorithmus bringen könnten.

Im Gegensatz zu den meisten anderen Techniken bzw. Parametern kann die Verwendung einer *Simulation Policy* keine Verbesserung der Spielstärke bewirken. Lediglich die Konfiguration (0,4 / 0,1) kann ähnlich stark wie der Algorithmus ohne Verwendung einer *Simulation Policy* spielen. Vermutlich kann der Nachteil durch den Mehraufwand der Zugsortierung nicht durch eine verbesserte Simulation ausgeglichen werden. Zu überprüfen ist, ob eine *Simulation Policy*, deren Zugsortierung auf der Bewertungsfunktion beruht, zu vielversprechenderen Ergebnissen führen kann. Zwar gibt es durch die Bewertungsfunktion einen höheren Rechenaufwand. Allerdings haben auch die Ergebnisse der Zugsortierung gezeigt (siehe Abb. 5.3), dass das Sortieren mithilfe der Bewertungsfunktion stärker ist.

Abalone hat einen hohen Verzweigungsgrad und dabei gleichzeitig viele schwache und ähnliche Züge („Geschwisterzüge“). Diesem Problem kann erfolgreich mit der Reduzierung des Verzweigungsgrades unter Beachtung von vorrangig guten Zügen (Verhältnis: (0,4 / 0,1)) entgegengewirkt werden.

Zuletzt ist zu erwähnen, dass die Wiederverwendung des Suchbaumes keine klaren Ergebnisse zeigt. Obwohl durch sie nicht schwächer gespielt wird, scheint sie auch keinen klaren Vorteil zu bieten. Mit einer in der Regel maximalen Suchbaumtiefe von drei ist anzunehmen, dass der Teilbaum nicht groß genug ist, um einen entscheidenen Vorteil zu bieten. Durch das Abändern von anderen Parametern könnte der Teilbaum größer und somit relevanter werden. Denkbar wäre zum Beispiel die Verringerung der Simulationstiefe, wodurch mehr Simulationen durchlaufen werden könnten, oder eine weitere Reduzierung des Verzweigungsgrades im Suchbaum.

Nichtsdestotrotz ist durch den systematischen Vergleich der verschiedenen Techniken eine klar spielstärkere Version des MCTS-Algorithmus entwickelt worden.

6.3. Einordnung der Ergebnisse des Algorithmenvergleichs

Die beiden Algorithmen konnten intern weiterentwickelt und damit deutlich verbesserte Versionen als die Basisalgorithmen erstellt werden. Bei dem direkten Vergleich ist aber eine deutliche Überlegenheit des Alpha-Beta-Algorithmus gegenüber dem MCTS-Algorithmus festzustellen. Wie schon in der Arbeit von Pascal Chorus [16] ist somit der heuristische Ansatz gegenüber dem stochastischen Ansatz im Vorteil.

Die Gründe für die deutliche Unterlegenheit des MCTS-Algorithmus könnte zum einem die geringe Anzahl an Simulationen (600 pro Suchdurchlauf) sein. Hier könnte unter anderem eine dynamische und somit vielleicht doch effizientere Berechnung des Spielzüge (siehe Kapitel 2.4) oder eine stärke Zuglimitierung in der Simulation Abhilfe schaffen.
Zum anderen ist in der Forschung schon bekannt, dass für einige Entscheidungsprobleme bzw. Spiele die Verwendung des MCTS-Algorithmus und für andere die der Alpha-Beta-Suche besonders geeig-

net ist.[6]
Ramanujan argumentiert in seiner Arbeit „On Adversarial Search Spaces and Sampling-Based Planning“ [18], dass die MCTS und der UCT-Ansatz bei Spielen mit sogenannten *trap states* schwach spielen. *Trap states* sind Spielsituationen, in denen die Ausführung eines nicht optimalen Zuges in nur wenigen darauf folgenden Spielzügen zu einer Niederlage führen kann.[18]
Abalone ist kein Spiel wie Schach, in dem effektiv nur eine Figur für den Sieg geschlagen werden muss und es daher zu einem “plötzlichen“ Spielende kommen kann. Dennoch tritt oft der Fall auf, dass Spielsituationen ausgeglichen erscheinen, aber mit wenigen Spielzüge ein unvermeidlicher Verlust der eigenen oder gegnerischen Kugel erzeugt werden kann. Ob diese Eigenschaft dazu führt, dass Abalone insgesamt nicht für den MCTS-Algorithmus geeignet ist, muss jedoch weiter untersucht werden.

6.4. Limitierungen der Arbeit

An dieser Stelle sei abschließend auf die Limitierungen der Arbeit und die daraus resultierenden Aspekte hingewiesen, die es gegebenenfalls weiter zu analysieren gilt.

Zunächst ist zu betonen, dass alle Parameter bzw. Techniken als unabhängig betrachtet werden. Dies geschieht vor allem zur Reduzierung der Komplexität. Techniken wie die Zuglimitierung in der Simulation und die *Simulation Policy* werden sich vermutlich durchaus beeinflussen. So ist anzunehmen, dass die *Simulation Policy* die Spiellänge eines randomisierten Spieles reduziert. Daraus resultiert vermutlich auch die Anpassung einer optimalen Simulationslimitierung. Solche gegenseitigen Einflüsse wurden nicht untersucht, sind aber wichtig zu betrachten.

Ebenfalls sind Parameter wie zum Beispiel die Größe und die Ersetzungsstrategie der *Transposition Table* ohne deren weitere Untersuchung festgelegt. Zuletzt ist auch zu beachten, dass es viele versteckte Parameter gibt, deren Einfluss erst durch weitere Analysen zum Vorschein kommen.

Viele Parameter sind zudem in einem gewissen zuvor ausgewählten Bereich untersucht worden. Beispielsweise sind der Wert der Fenstergröße des *Aspiration Window* oder die verschiedenen Zugverhältnisse bei der *Simulation Policy* und der Reduzierung des Verzweigungsgrades nur in einem vorher als geeignet angesehen Bereich untersucht worden.

Für eine fundiertere Einschätzung der Stärke der Bewertungsfunktion bedarf es ebenfalls weiterer Tests. In diesem Bereich ist es besonderes schwierig, Aussagen über die Stärke zu treffen. Zwar wurde ein Vergleich mit der Bewertungsfunktion aus der Arbeit von Pascal Chorus [16] gezogen, dennoch sind für weitere Analysen Spiele gegen andere Implementationen und auch menschliche Spieler nötig. Gerade das Spielen gegen besonders erfahrene menschliche Spieler kann konzeptionelle Schwächen aufzeigen.

Zuletzt wurde das Problem der Begrenztheit von Ressourcen nicht beleuchtet. So wurden der Speicherplatz und die verwendete Rechenleistung nicht analysiert und zudem nicht untersucht, wie sich knappe Ressourcen (Zeit, Speicher, Rechenleistung) auf die Leistung der Algorithmen auswirken.

7. Fazit

In der vorliegenden Arbeit wurden die Alpha-Beta-Suche und der MCTS-Algorithmus für das Spiel Abalone sowie weitere Techniken zur Verbesserung der Spielverhalten entwickelt. In einem systematischen Vergleich wurden die Parameter und Techniken der Algorithmen ausgewertet.

Die Ergebnisse der Alpha-Beta-Suche können überwiegend die bestehenden Forschungsergebnisse aus der Literatur bestätigen. Dennoch sind auch Unterschiede, wie zum Beispiel bei der Ruhesuche, festzustellen.
Für den MCTS-Algorithmus sind zudem neuen Erkenntnisse gewonnen worden. So scheint eine *Simulation Policy* wenig Verbesserung zu bewirken. Dagegen hat die Reduzierung des Verzweigungsgrades im Suchbaum positive Auswirken auf die Spielstärke des Algorithmus.
Insgesamt sind beide Algorithmen weiterentwickelt worden. Bei dem direkten Vergleich stellt sich aber die Alpha-Beta-Suche als deutlich stärker heraus. Es bedarf weiterer Untersuchungen, ob durch Einführung neuer Techniken beim MCTS-Algorithmus eine KI entwickelbar ist, die eine auf der Aplha-Beta-Suche basierende KI im Spiel Abalone schlagen kann.

In zukünftigen Arbeiten gilt es unter anderem zu untersuchen, inwieweit die einzelnen Methoden verbessert werden können. Als vielversprechender Ansatz könnte sich hier eine Spielphasenerkennung herausstellen. Diese könnte sowohl bei der Bewertungsfunktion und Zugsortierung in der Alpha-Beta-Suche als auch bei der Zuglimitierung des MCTS-Algortihmus Verbesserungen mit sich bringen. Auch eine Kombination aus der MCTS und anschließender Alpha-Beta-Suche könnte erfolgreich sein.

Abschließend ist festzuhalten, dass die Alpha-Beta-Suche ein erfolgreicher Algorithmus für die Zugauswahl im Spiel Abalone ist. Für den MCTS-Algorithmus wurden in dieser Arbeit nur die Grundsteine gelegt, weshalb hier für die Entwicklung einer starken künstlichen Intelligenz eine tiefgreifendere Forschung nötig ist.

Literaturverzeichnis

[1] Oswin Aichholzer, Franz Aurenhammer, and Tino Werner. Algorithmic fun-abalone. *Special Issue on Foundations of Information Processing of TELEMATIK*, 1:4–6, 2002.

[2] Ariyurek, Sinan and Betin-Can, Aysu and Surer, Elif. Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing. In *2020 IEEE Conference on Games (CoG)*, pages 25–32, 2020. https://ieeexplore.ieee.org/document/9231670.

[3] Benjamin Blankertz und Vera Röhr. Algorithmen und Datenstrukturen. https://git.tu-berlin.de/algodat-sose21/Material/-/blob/master/LectureNotes_AlgoDat.pdf, 2021. [Online; Stand 15. Februar 2024].

[4] Breuker, Dennis. Memory versus search in games. 21:245–247, 12 1998. https://www.researchgate.net/publication/298961626_Memory_versus_search_in_games.

[5] Breuker, Dennis and Uiterwijk, Jos and Herik, H. Replacement Schemes for Transposition Tables. *ICCA Journal*, 17, 02 1970. https://www.researchgate.net/publication/2737817_Replacement_Schemes_for_Transposition_Tables.

[6] Browne, Cameron B. and Powley, Edward and Whitehouse, Daniel and Lucas, Simon M. and Cowling, Peter I. and Rohlfshagen, Philipp and Tavener, Stephen and Perez, Diego and Samothrakis, Spyridon and Colton, Simon. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6145622.

[7] Coulom, Rémi. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 72–83, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Don F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990. https://www.sciencedirect.com/science/article/pii/0004370290900728.

[9] Dr. Christian Meilicke. Monte Carlo Tree Search. https://web.informatik.uni-mannheim.de/cmeilicke/grundlagen/KIX-06-MCTS.pdf. [Online; Stand 21. Februar 2024].

[10] Florian Lemmerich und Bastian Spaeth. Suchoptimierung und Evaluation bei Zwei-Personen-Nullsummen-Spielen am Beispiel von Muhle. https://www.informatik.uni-wuerzburg.de/fileadmin/10030600/Anwendungen/Muehle/Muehle_Lemmerich-Spaeth.pdf, 2005. [Online; Stand 12. Februar 2024].

[11] Hikari Kato, Szilárd Zsolt Fazekas, Mayumi Takaya, and Akihiro Yamamura. Comparative study of monte-carlo tree search and alpha-beta pruning in amazons. In Ismail Khalil, Erich Neuhold, A Min Tjoa, Li Da Xu, and Ilsun You, editors, *Information and Communication Technology*, pages 139–148, Cham, 2015. Springer International Publishing.

[12] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. https://www.sciencedirect.com/science/article/pii/0004370285900840.

[13] Michel Lalet und Laurent Lévi. abalone, spielregeln. https://www.spielezar.ch/modules/genzo_zar/views/pdf/spielregeln-abalone-classic.pdf, 2014. [Online; Stand 15. März 2024].

[14] N.P.P.M. Lemmens. Constructing an Abalone Game-Playing Agent. https://project.dke.maastrichtuniversity.nl/games/files/bsc/Lemmens_BSc-paper.pdf, 2005. [Online; Stand 12. Februar 2024].

[15] Papadopoulos, Athanasios and Toumpas, Konstantinos and Chrysopoulos, Antonios and Mitkas, Pericles A. *Exploring optimization strategies in board game Abalone for Alpha-Beta search*. 2012. https://ieeexplore.ieee.org/abstract/document/6374139.

[16] Pascal Chorus. Implementing a Computer Player for Abalone using Alpha-Beta and Monte-Carlo Search. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=99b550ef0acea8a92392691a6598315867c0573b, 2009. [Online; Stand 12. Februar 2024].

[17] Prof. Dr. Bernhard Nebel. ISpieltheorie. https://gki.informatik.uni-freiburg.de/teaching/ss09/gametheory/spieltheorie.pdf, 2009. [Online; Stand 21. März 2024].

[18] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On Adversarial Search Spaces and Sampling-Based Planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 20(1):242–245, May 2021.

[19] Michiel Verloop. A Critical Review: Exploring Optimization Strategies in Board Game Abalone for Alpha-Beta Search. https://www.cs.ru.nl/bachelors-theses/2021/Michiel_Verloop___1009995___A_Critical_Review_-_Exploring_Optimization_Strategies_in_Board_Game_Abalone_for_Alpha-Beta_Search.pdf#page=40&zoom=100,157,390. [Online; Stand 12. Februar 2024].

[20] Wikipedia. Abalone (Spiel) — Wikipedia, die freie Enzyklopädie. https://de.wikipedia.org/w/index.php?title=Abalone_(Spiel)&oldid=240985062, 2024. [Online; Stand 15. März 2024].

[21] Wikipedia contributors. Abalone (board game) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Abalone_(board_game)&oldid=1182526146, 2023. [Online; Stand 15. März 2024].

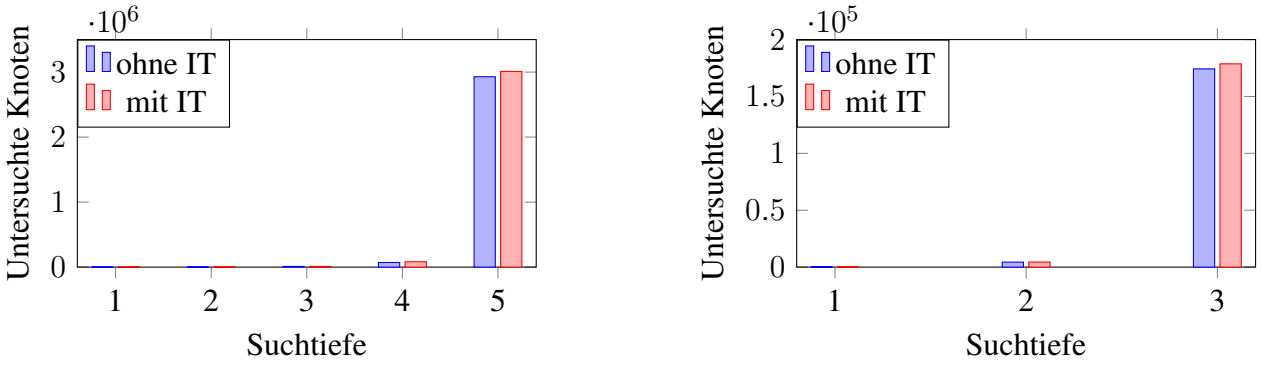
[22] Świechowski, Maciej and Godlewski, Konrad and Sawicki, Bartosz and Mańdziuk, Jacek. Monte Carlo Tree Search: a review of recent modifications and applications. 2023. https://link.springer.com/article/10.1007/s10462-022-10228-y#citeas.

A. Anhang

```
#!/bin/bash
#SBATCH --job-name=ba_test_malte_hauff
#SBATCH --partition=cpu-5h
#SBATCH --gpus-per-node=0
#SBATCH --ntasks-per-node=10
#SBATCH --output=logs/job-%j.out

apptainer run python_container.sif python src/main.py
```

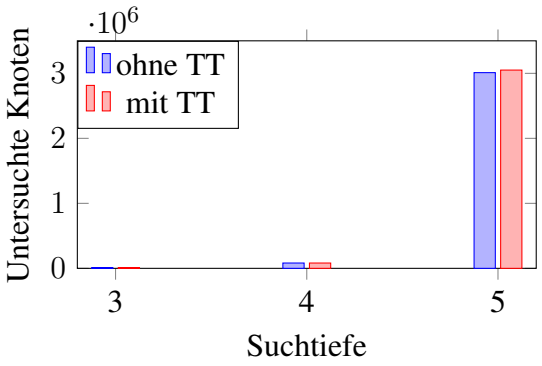
Abbildung A.1.: Jobkonfiguration für das Cluster



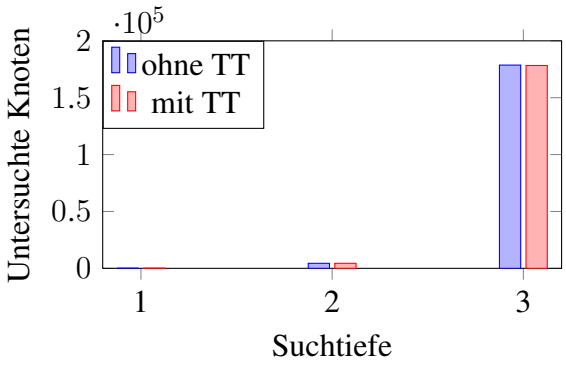
(a) Belgium-Daisy-Aufstellung

(b) Mittelspielstellung 2

Abbildung A.2.: Vergleich 2 mit und ohne iterative Tiefensuche (IT)

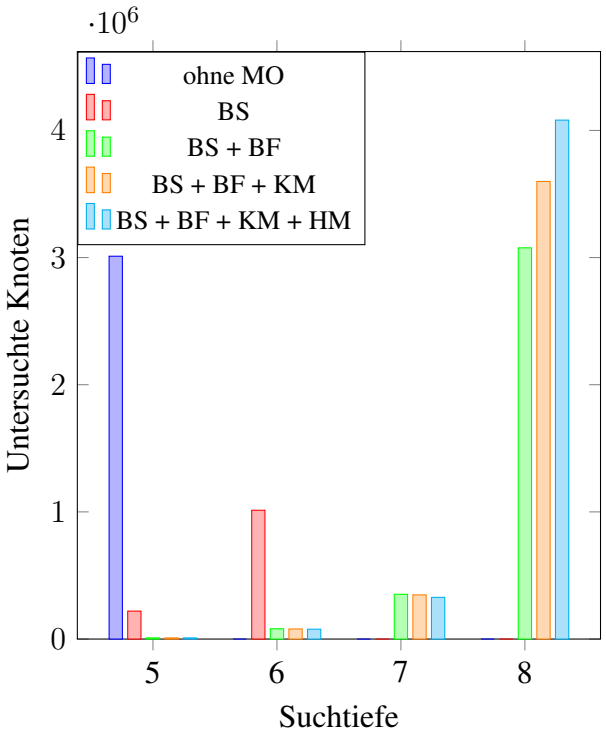


(a) Belgium-Daisy-Aufstellung

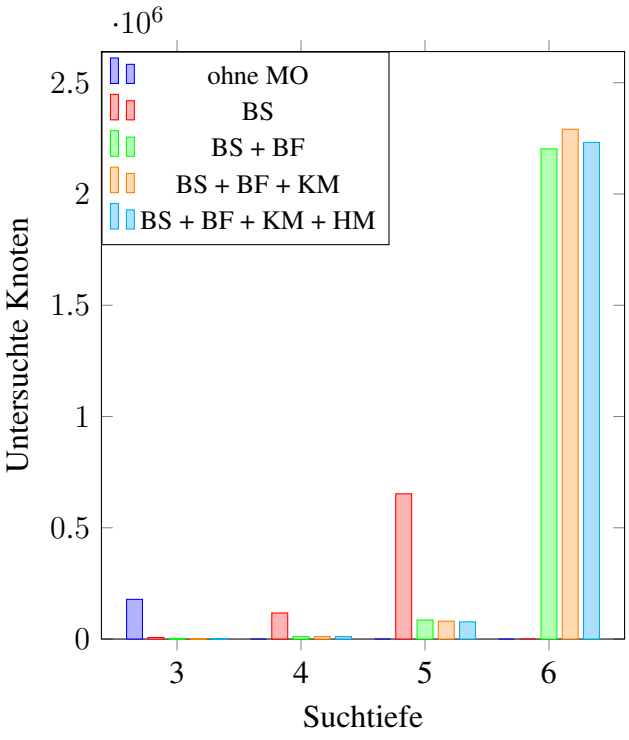


(b) Mittelspielstellung 2

Abbildung A.3.: Vergleich 2 mit und ohne *Transposition Table* (TT)

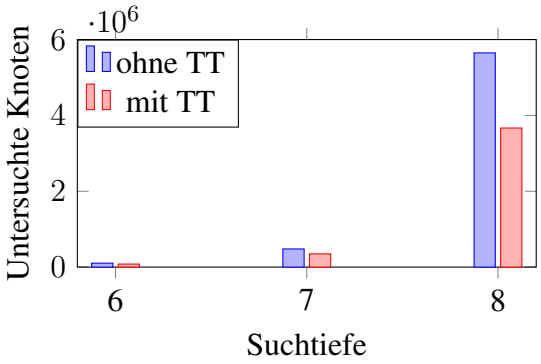


(a) Belgium-Daisy-Aufstellung

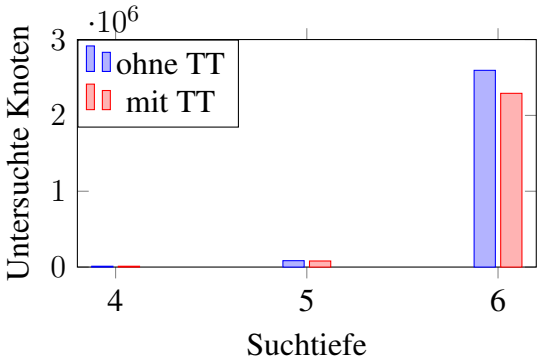


(b) Mittelspielstellung 2

Abbildung A.4.: Vergleich 2 der Zugsortierung (move ordering, MO)

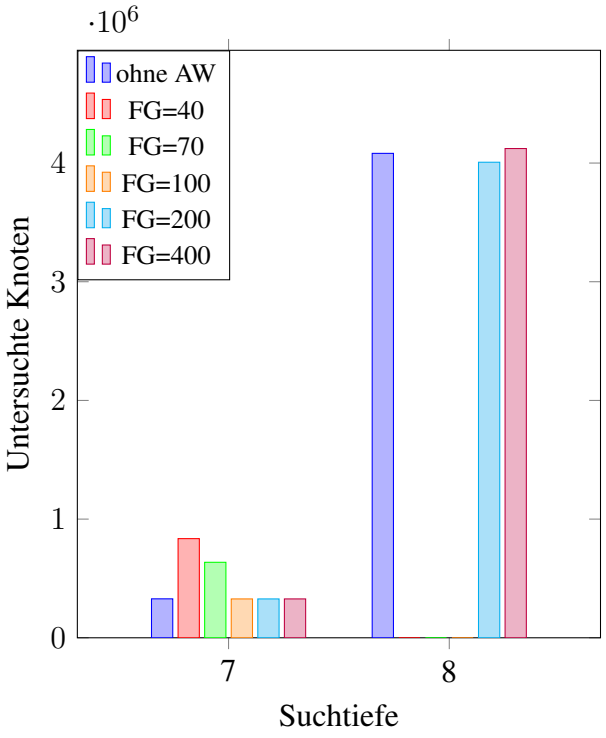


(a) Belgium-Daisy-Aufstellung

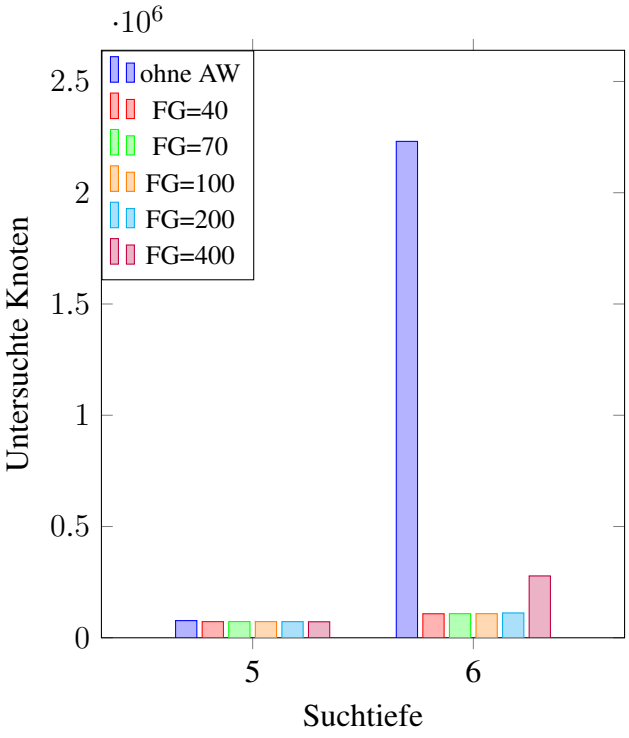


(b) Mittelspielstellung 2

Abbildung A.5.: Vergleich 2 mit und ohne *Transposition Table* (TT) und Zugsortierung (BS + BF + KM)



(a) Belgium-Daisy-Aufstellung



(b) Mittelspielstellung 2

Abbildung A.6.: Vergleich 2 mit und ohne *Aspiration Window* (AW) für verschiedene Fenstergrößen (FG)