DHBW
Duale Hochschule
Baden-Württemberg

Quiescence Search

Hausarbeit

des Studiengangs Informatik, Vorlesung Wissensbasierte Systeme

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Florian Redmann und Felix Wortmann

Matrikelnummer, Kurs 1793766 und 3287310, TINF18-AI2

Ausbildungsfirma REWE Systems GmbH und DB Systel GmbH

Inhaltsverzeichnis

1 Einleitung

Diese Hausarbeit beschäftigt sich mit dem Thema *Quiescence Seach* oder auch *Ruhesuche*. Sie wurde im Kontext der Vorlesung „Wissensbasierte Systeme“, gehalten an der DHBW Mannheim von Prof. Dr. Karl Stroetmann, erstellt. Diese Arbeit sowie die Implementierungen – alle relevanten Teile werden nichtsdestotrotz in dieser Arbeit dargestellt – lassen sich auf *GitHub* unter folgendem Link abrufen: https://github.com/felixwortmann/quiescence_search.

1.1 Aufgabenstellung und -kontext

Ziel dieser Arbeit ist es, das Verfahren der Quiescence Search zu erläutern und eine Implementierung im Rahmen einer Schach-KI (Künstliche Intelligenz) zu demonstrieren. Diese KI wurde im Rahmen einer Studienarbeit, ebenfalls betreut von Prof. Dr. Karl Stroetmann, entwickelt und basiert auf dem *Minimax-Algorithmus* sowie auf dem *Alpha-Beta-Pruning* und der *Memoization*. Bevor auf den Quiescence Search Algorithmus eingegangen wird, werden jene Implementierungen gezeigt und erläutert sowie die Theorie, auf der sie aufbauen, erklärt. Schließlich wird auf die vorhandene KI die Quiescence Search aufgebaut und implementiert, sodass als Resultat dieser Arbeit ein funktionierender Algorithmus entsteht, welcher mithilfe von Quiescence Search eine Verbesserung zu dem bestehenden darstellt.

2 Der Minimax-Algorithmus

In diesem Kapitel wird erläutert, wie die bestehende Implementierung der Schach-KI aufgebaut ist und wie diese funktioniert. Die vorgestellten Implementierungen können ebenfalls auf *GitHub* unter https://github.com/felixwortmann/chess abgerufen werden.

2.1 Das Schach-Spiel

Zunächst werden die Grundlagen des Schach-Spiels erläutert. Es handelt sich bei Schach um ein Brettspiel für zwei Spieler, wobei einer der Spieler mit der Farbe Weiß, der andere mit der Farbe Schwarz spielt. Jeder der Spieler hat jeweils 16 Figuren, davon acht Bauern, jeweils zwei Springer, Läufer und Türme, eine Dame sowie einen König. Der Spieler der weißen Farbe beginnt stets, wobei jeder Spieler die folgenden Züge mit den jeweiligen Figuren ausführen kann:

- Bauer – Die schwächste Figur im Spiel, kann pro Zug ein Feld nach vorne rücken (falls es der erste Zug des Bauern ist, kann dieser ein oder zwei Felder vorrücken) und gegnerische Figuren schlagen, falls sie diagonal ein Feld vor ihm liegen
- Springer – Die einzige Figur im Spiel, die über andere Figuren „springen“ kann. Das Zugmuster des Springers gleicht einem L, da er zwei Felder in eine Richtung, dann ein Feld orthogonal zur Seite ziehen kann. Die Figuren, die er überspringt, werden nicht beeinflusst
- Läufer – Der Läufer kann sich auf einer Diagonale in alle Richtungen bewegen
- Turm – Türme können sich ähnlich den Läufern auf einer Linie bewegen, jedoch nicht diagonal sondern horizontal und vertikal
- Dame – Die Dame hat die meisten Möglichkeite zum Zug: diagonal, horizontal, vertikal sowie alle umliegenden Felder
- König – Der König kann sich zwar nur pro Zug ein Feld in eine beliebige Richtung bewegen, ist aber insofern die wichtigste Figur des Spiels, als dass er über Niederlage eines Spieler unterscheiden kann: Steht eine gegnerische Figur so, dass sie im nächsten Zug den König schlagen könnte, ist er *schach* gesetzt; hat der Spieler zusätzlich keine Mögichkeit, den König zu schützen (durch Zug des Königs auf ein anderes Feld,

Blockieren der Angriffslinie durch eine andere Figur oder schlagen der bedrohenden Figur), spricht man von *Schachmatt* und der gegnerische Spieler hat gewonnen

Eine Möglichkeit, wie das Spiel enden kann, wurde bereits erwähnt: das *Schachmatt*. Bei Schach handelt es sich um ein sogenanntes *Nullsummenspiel*, sodass, analog zu Spielen wie beispielsweise *Tic Tac Toe* oder *Vier gewinnt*, das Spiel nur auf drei Weisen enden kann: Spieler A (weiß) gewinnt, Spieler B (schwarz) gewinnt oder es herrscht Gleichstand, *Remis* genannt.

Die drei Möglichen stellen sich folgendermaßen dar:

1. Weiß gewinnt: Schwarz wurde schachmatt gesetzt oder hat aufgegeben
2. Schwarz gewinnt: Weiß wurde schachmatt gesetzt oder hat aufgegeben
3. Remis: Spieler einigen sich auf ein Remis, einer der Spieler steht *patt* (keine möglichen Züge, der König ist jedoch nicht schachmatt gesetzt), beide Spieler haben nicht genug Figuren, um das Spiel zu gewinnen oder, die letzte Variante, eine bestimmte Anzahl an Zügen wurde (wiederholt) ausgeführt

2.2 Grundgedanke einer Schach-KI

Der grundlegende Ansatz einer Schach-KI besteht darin, dass das Programm alle möglichen Züge analysiert und den besten Zug auswählt. Aufgrund der oben erwähnten Anzahl an möglichen Zügen für jeden Spieler, wächst die Zahl der möglichen Spiele, die daraus resultieren, jedoch stark an. Es wäre daher de facto unmöglich, *alle* Züge einer kompletten Schach-Partie zu analysieren, da kein Computer der Welt je genug Speicherplatz haben würde, um alle möglichen Ergebnisse abzuspeichern: Die Anwendung der Rechnung von *Claude Shannon* ergibt eine geschätzte Untergrenze von 10^{120} möglichen Spielpositionen in einer gesamten Schachpartie[1]; die Anzahl der Atome im beobachtbaren Universum beträgt jedoch 10^{80}, Schätzungen zufolge[2]. Somit wird es, unabhängig von der Rechenleistung, nie möglich sein, alle unterschiedlichen Spiel-Ausgänge abzuspeichern.

Entscheidet man sich nun, statt aller möglichen Züge, nur bis zu einer bestimmten Tiefe zu suchen, verringert sich die Anzahl an möglichen Zügen zwar, bleibt aber dennoch zu hoch, da sich die Anzahl an möglichen Spiel-Zuständen nach Claude Shannon mit jedem vollen Zug um den Faktor 10^3 erhöht[3]. Es benötigt also einen Algorithmus, welcher effizient durch eine begrenzte Anzahl möglicher Züge geht und analysiert, welcher dieser Züge die

[1] Shannon 1950.
[2] Merz 2002.
[3] Shannon 1950.

beste Folgeposition zur Folge hat. Der genutzte Algorithmus ist der *Minimax-Algorithmus*, genauer die Variante *Negamax*, welche jedoch de facto analog zu dem Standard-Minimax-Algorithmus funktioniert und lediglich die Rückgabewerte so anpasst, dass beide Spieler maximierend sind, statt der Standadvariante in der der Spieler der Farbe Weiß maximierend und der Spieler der Farbe Schwarz minimierend ist – dies ist auch der Grund für die Namensgebung des Algorithmus.

2.3 Funktionsweise des Minimax-Algorithmus

Der Minimax-Algorithmus funktioniert *rekursiv*, das heißt, dass die Funktion sich selbst so lange aufruft, bis die Abbruchbedingung erreicht wird. Die Abbruchbedingungen in diesem Kontext sind, dass entweder

1. die maximale Suchtiefe erreicht wird oder
2. keine weiteren Züge zur Analyse vorhanden sind.

Abbildung 2.1 stellt den Ablauf des Algorithmus dar. Zu sehen ist ein *Baum*, welcher die verschiedenen Zustände durch Zahlen abbildet. Diese Zahlen können sowohl negativ als auch positiv sein; die negativen Zahlen sind dabei gut für den minimierenden Spieler (Schwarz), während positive Zahlen gut für den maximierenden Spieler (Weiß) sind. Außerdem sind die Werte $+\infty$ und $-\infty$ zu sehen – diese Werte stellen eine Schachmatt-Position dar, können jedoch auch anders implementiert werden. Der Algorithmus geht so vor, dass er rekursiv durch den abgebildeten Baum iteriert und den Pfad raussucht, welcher zum besten Ergebnis (in der Abbildung wäre dies der Pfad, der zu $+\infty$ führt) führt.

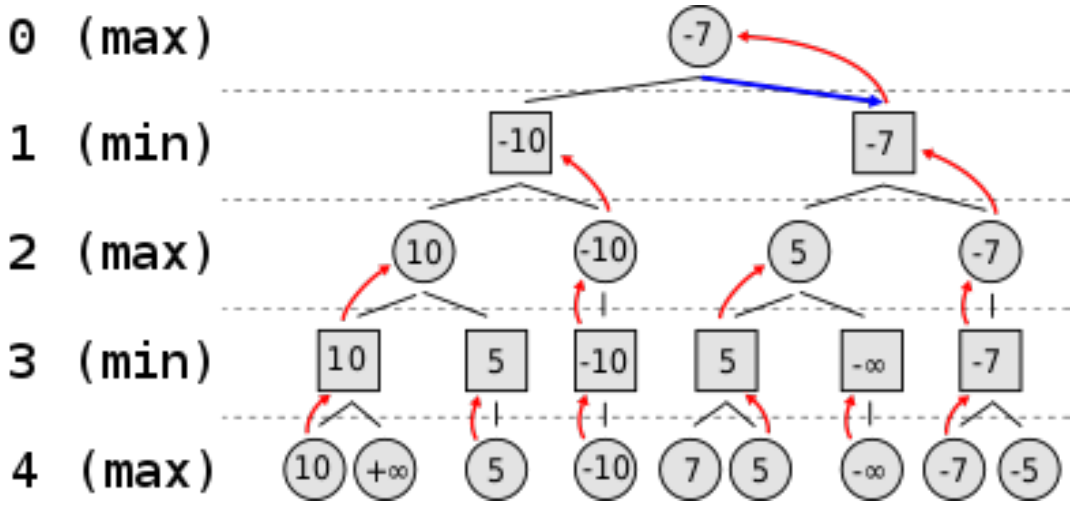

Abbildung 2.1: Überblick über die Vorgehensweise des Minimax-Algorithmus[1]

Die Werte werden mithilfe einer weiteren Funktion berechnet, indem verschiedene Faktoren in die Berechnung miteinbezogen werden. Beispiele für solche Faktoren sind:

- Die Anzahl der jeweiligen Figuren, die verschiedenen Figurtypen haben dabei verschiedene Werte: Eine Dame entspricht dabei beispielsweise dem Wert von neun Bauern
- Die Position der einzelnen Figuren. Ein Springer im Zentrum ist wertvoller als ein Springer am Rand, da er im Zentrum mehr Zugmöglichkeiten hat
- Die Tatsache, ob ein Schachmatt herrscht

Die Verwendung des Minimax-Algorithmus allein wird zwar eine intelligente Berechnung der Züge erlauben, jedoch, aufgrund der weiterhin großen Anzahl an möglichen Zügen, keine hohe Suchtiefe ermöglichen. Um die Berechnungsdauer zu verringern – und somit eine hohere Suchtiefe zu ermöglichen – bestehen daher Verbesserungsmöglichkeiten. Die implementierten Verbesserungen sind dabei das sogenannte *Alpha-Beta-Pruning*, die Verwendung einer *Transpositionstabelle* (auch als *Memoization* oder *Caching* bezeichnet) sowie die Verwendung einer *Eröffnungsbibliothek*, welche erlaubt, zu Beginn der Partie jene Züge, die häufig bei Eröffnungen gespielt werden, direkt kontern zu können, ohne den Algorithmus zu benutzen.

Die Verwendung einer Transpositionstabelle bedeutet schlicht, dass bereits berechnete Züge in einem Cache gespeichert werden. Trifft der Algorithmus so zu einem späteren Zeitpunkt auf eine Position, die bereits ausgerechnet wurde, kann der Wert aus dem Cache geladen werden, um eine unnötige erneute Berechnung zu verhindern. Wird ein solches Caching genutzt, bietet sich das erwähnte Alpha-Beta-Pruning an. Abbildung 2.2 stellt den groben Ablauf bei der Verwendung von Alpha-Beta-Pruning dar.

[1] Wikipedia 2021b

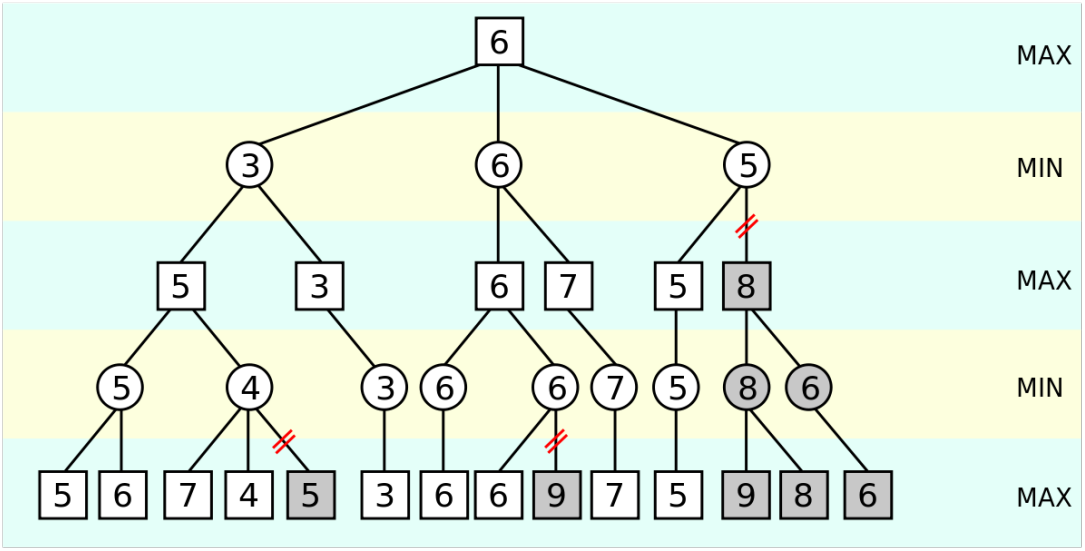

Abbildung 2.2: Überblick über die Vorgehensweise des Alpha-Beta-Prunings[1]

Genauer erklärt, werden für diese Erweiterung des Algorithmus zwei weitere Werte verwendet: *Alpha* und *Beta*. Diese beiden Werte stellen zwei Grenzen dar, welche als Worst-Case-Szenarios gesehen werden können. Der Wert *Alpha* stellt die untere Grenze und somit das Worst-Case-Szenario für die KI dar, während *Beta* die obere Grenze und somit das Worst-Case-Szenario für den menschlichen Spieler abbildet. Der Algorithmus läuft so ab, dass mithilfe der Schranken die Werte der Zustands-Evaluation überprüft werden; überschreitet ein Wert die *Beta*-Grenze, so wird der gesamte Teilbaum von diesem Knoten ausgehend „abgeschnitten“ – ergo nicht weiter betrachtet – da die gegnerische Seite (der menschliche Spieler) diesen Teilbaum nicht betreten wird, weil die bisher evaluierten Teilbäume bereits bessere Ergebnisse für den Spielern liefern. Die *Alpha*-Grenze wird so verwendet, dass die Werte der Zustands-Evaluation nur dann übernommen – und somit als neuer *Alpha*-Wert gesetzt werden – werden, falls sie die die bestehende *Alpha*-Grenze nicht unterschreiten, da in diesem Fall der betrachtete Zug schlechter als der aktuell beste Zug wäre.

2.4 Implementierung

In diesem Abschnitt wird demonstriert, wie der oben erwähnte Algorithmus praktisch umgesetzt wurde. Es wird nicht der vollständige Code gezeigt, sondern lediglich die Haupt-Funktion des Minimax-Algorithmus, da sich diese Arbeit weniger auf den bereits vorhandenen Algorithmus fokussiert, sondern mehr auf die Implementierung der Quiescence Search. In Kapitel 3 hingegen wird der gesamte Code gezeigt, der im Rahmen der

[1] Wikipedia 2021a

Implementierung geschrieben wurde. Es sei an dieser Stelle nichtsdestotrotz erneut auf https://github.com/felixwortmann/chess verwiesen, wo der gesamte Quellcode für die Schach-KI sowie diverse Hilfs- und Testfunktionen abrufbar ist.

```
@memoize_minimax
def minimax(board, depth, alpha, beta):
    global BEST_MOVE
    global MINIMAX_CALLS
    MINIMAX_CALLS += 1
    # Check, if current board is in cache
    # If the depth is zero, give the static evaluation of the current
        board and save it in the cache
    if depth == 0 or not board.legal_moves:
        value = static_eval(board, is_endgame(board))
        return value
    max_value = alpha
    ordered_moves = []
    cnt = 0
    # Order the moves roughly, using a static evaluation for every move
    for move in board.legal_moves:
        cnt += 1
        board.push(move)
        v = static_eval(board, is_endgame(board))
        board.pop()
        heapq.heappush(ordered_moves, (v, cnt, move))
    # Calculate the minimax value recursively, using alpha-beta-pruning
    for _, _, move in ordered_moves:
        board.push(move)
        value = -minimax(board, depth - 1, -beta, -max_value)
        board.pop()
        if value > max_value:
            max_value = value
            if depth == ANALYZING_DEPTH:
                BEST_MOVE = move
            if max_value >= beta:
                break
    return max_value
```

Listing 2.1: Funktion für den Minimax-Algorithmus mit Alpha-Beta-Pruning

Der Code-Ausschnitt 2.1 zeigt die Funktion, welche den Minimax-Algorithmus implementiert. Es sei erneut erwähnt, dass es sich technisch gesehen um einen Negamax-Algorithmus handelt, weshalb in der Funktion keine Unterscheidung zwischen maximierendem und minimierendem Spieler gemacht wird. Stattdessen werden die Werte bei Aufruf der Funktion sowie der zurückgegebene Wert des rekursiven Aufrufs invertiert.

Die Funktion wird mit den folgenden Parametern aufgerufen:

- `board`, der Zustand des aktuellen Spielbrettes. Für das Brett wurde die Python-Bibliothek *python-chess* verwendet
- `depth`, zu Beginn die gewünschte Suchtiefe (standardmäßig beträgt diese 5), danach verringert sich der Wert mit jedem rekursiven Aufruf
- `alpha` und `beta`, die oben erwähnten Werte, welche die Schranken für das Alpha-Beta-Pruning darstellen

Der Ablauf der Funktion lässt sich folgendermaßen zusammenfassen:

1. Zunächst wird geprüft, ob die Tiefe 0 beträgt, die Funktion also „unten“ am Baum angekommen ist, oder keine weiteren Züge zur Analyse vorhanden sind. In diesem Fall wird die Auswertung der aktuellen Spielposition zurückgegeben
2. Anschließend, falls Punkt 1 übersprungen wurde, werden die vorhandenen Züge grob sortiert, indem auf sie die Evaluations-Funktion angewendet wird und sie anschließend anhand dessen sortiert werden
3. Nun wird durch die sortierten Züge iteriert und die Funktion ruft sich rekursiv selbst auf. Der Rückgabewert wird invertiert (aufgrund der Negamax-Variante) und das Alpha-Beta-Pruning wird angewendet

Die Sortierung der Züge verbessert die Performanz der Schach-KI zusätzlich, da das Alpha-Beta-Pruning so deutlich effektiver angewendet werden kann.

3 Quiescence Search

Dieses Kapitel beschäftigt sich mit der konkreten Zielsetzung dieser Arbeit: Eine Ergänzung der Schach-KI, sodass die Quiescence Search implementiert ist und das Programm somit effektivere Züge berechnen kann.

3.1 Motivation

Zunächst gilt zu klären, welche Motivation beziehungsweise welche Problemstellung der Notwendigkeit der Quiescence Search zugrunde liegt. Die Bezeichnung für das Problem, welches bei der vorhandenen Implementierung des Algorithmus auftritt, lautet *Horizonteffekt*. Aufgrund der Tatsache, dass der Algorithmus mit einer festen Tiefe arbeitet, existiert für das Programm ein metaphorischer Horizont, über welchen es nicht hinausschauen kann. *Oxford Reference* definiert den Begriff *Horizonteffekt* folgendermaßen: „*The horizon effect refers to the fact that interesting results will always exist beyond any depth D and therefore in any given search will not be discovered [...]*“[1].

Abbildung 3.1 zeigt eine beispielhafte Aufstellung der Figuren auf einem Schachbrett. Aufgabe der KI ist es nun, den bestmöglichen Zug zu finden[2]. Das Programm wird schnell erkennen, dass die Dame auf B2 den Springer auf B7 schlagen kann. Da der Horizont an dieser Stelle „endet“, schätzt das Programm eben diesen Zug als den besten ein und gibt ihn zurück. Wäre die Suchtiefe jedoch um eine Ebene tiefer gewesen, hätte das Programm diesen Zug ausgeschlossen, da, nachdem die Dame den Springer geschlagen hat, der Läufer auf A8 die Dame direkt schlagen kann. Durch die Tatsache, dass eine Dame jedoch deutlich mehr wert ist als ein Springer – in der vorhandenen Implementierung ist eine Dame 9 Bauern wert, ein Springer jedoch nur 3,2 Bauern – handelt es sich um einen sehr schlechten Zug.

[1] Oxford Reference 2021.

[2] Um das Beispiel möglichst einfach zu halten, wird angenommen, dass die KI mit einer Tiefe von 1 rechnet. Für die Erklärung der Problemstellung macht dies jedoch keinen Unterschied, da sich das Beispiel übertragen lässt

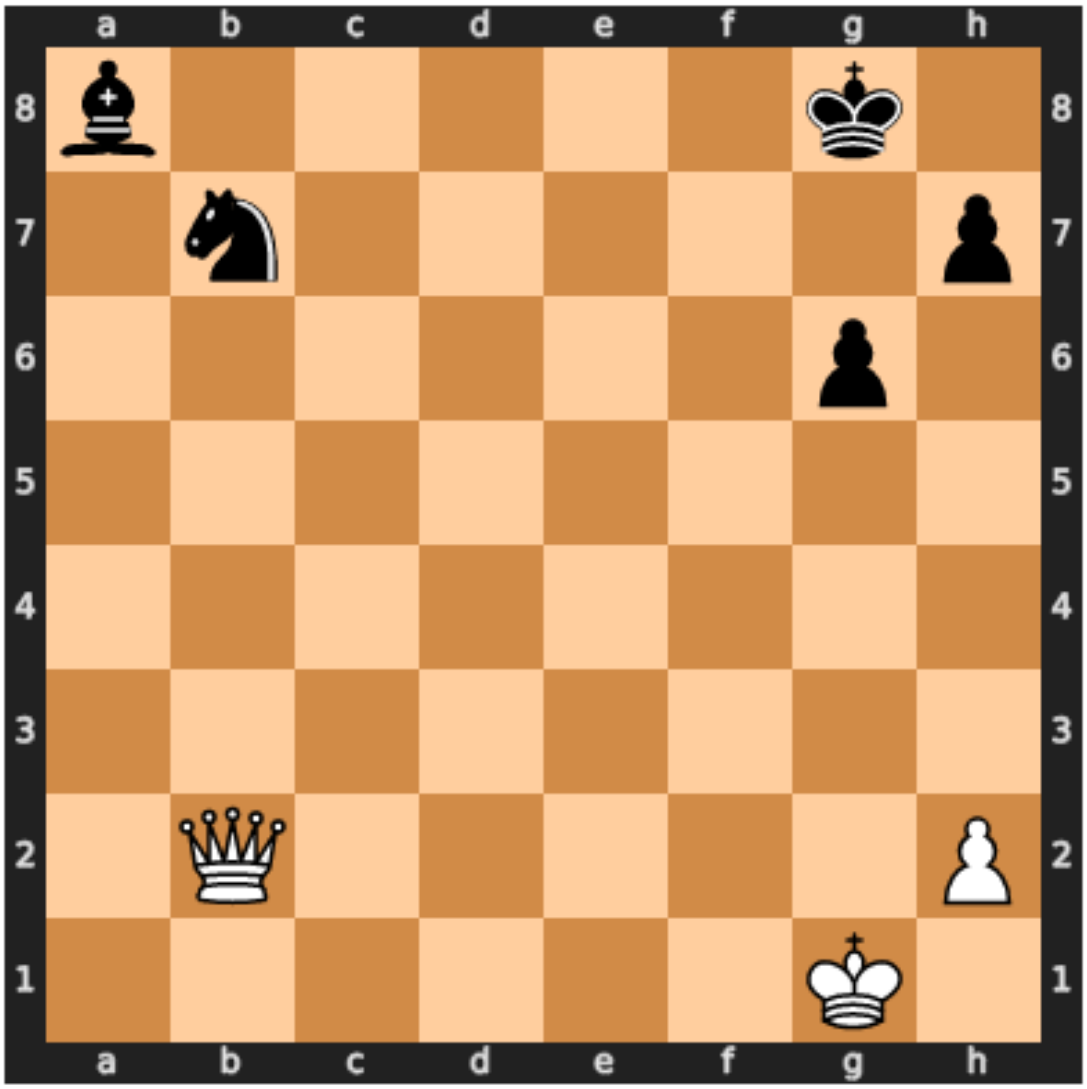

Abbildung 3.1: Beispiel-Zustand für ein Schachbrett: Weiß kann im nächsten Zug einen Springer schlagen

Ein weiteres Beispiel für den Horizonteffekt ist die scheinbare Vermeidung von schlechten Positionen: Das Programm erkennt eine schlechte Position, beispielsweise den Verlust einer Dame. Nun errechnet die KI einen Pfad, in dem diese Dame scheinbar nicht verloren wird; in der Realität hat das Programm den Verlust jedoch nur „über“ den Horizont geschoben, sodass der Verlust der Dame unausweichlich ist – zusätzlich hat das Verschieben des Problems womöglich eine insgesamt schlechtere Position hervorgerufen, als wenn die Dame direkt geopfert worden wäre.

Diese Beispiele verdeutlichen, dass, bei taktischen Zügen, eine tiefere Evaluation nicht nur sinnvoll sondern auch notwendig ist, um die Effektivität der Schach-KI zu steigern. Wird bei der tieferen Evaluation festgestellt, dass der Zug keine deutlich schlechtere Position hinter dem Horizont hervorruft, spricht man von einer *ruhigen* Stellung, was auch die Bezeichnung der *Ruhesuche* erklärt.

3.2 Funktionsweise

Bevor die Quiescence konkret implementiert werden kann, muss geklärt werden, wie sie funktioniert und wann sie eingesetzt wird.

In Abschnitt 3.1 wurden *taktische Züge* erwähnt. Das *Chess Programming Wiki* definiert den Begriff folgendermaßen: „*Tactical moves in the context of chess program move classification are moves which immediate change material balance and capture a piece [...]*“[1]. Es handelt sich also bei taktischen Züge um das Schlagen von Figuren respektive um das Aufwerten eines Bauern. Die Erweiterung des Algorithmus muss also dann eine tiefere Suche durchführen, wenn ein solcher taktischer Zug der zuletzt analysierte war.

Die Position in der Implementierung, an der die Quiescence Search eingesetzt werden muss, lässt sich ebenfalls definieren: So muss die Funktion für die Quiescence Search dann aufgerufen werden, wenn das Programm an der Suchtiefe angekommen ist. Innerhalb der Funktion wird dann geprüft, ob der letzte Zug ein taktischer Zug war, um eventuell tiefer zu suchen. Die tiefere Suche wird lediglich durch einen erneuten Aufruf der Minimax-Funktion umgesetzt.

3.3 Implementierung

Dieser Abschnitt zeigt die Implementierung des Quiescence Search Algorithmus. Außerdem wird erläutert, wie die korrekte Funktionsweise der Quiescence Search geprüft wurde.

```
1  @memoize_minimax
2  def minimax(board, depth, alpha, beta, is_quiesce):
3    # ...
4    if depth == 0 or not board.legal_moves:
5      value = static_eval(board, is_endgame(board))
6      if is_quiesce:
7        return value
8      return quiesce(board,alpha,beta,value)
9    # ...
```

Listing 3.1: Erweiterung der Funktion des Minimax-Algorithmus für die Verwendung von Quiescence Search

Der Code-Ausschnitt 3.1 zeigt den Teil der Minimax-Funktion, der für die Verwendung der Quiescence Search erweitert wurde. Zum einen wurde der Funktion der Parameter `is_quiesce` hinzugefügt (dieser wird bei der Erläuterung der Funktion `quiesce` genauer

[1] Chess Programming Wiki 2021.

erklärt), zum anderen wurde der Aufruf der Funktion `quiesce` mit den Parametern `board`, `alpha`, `beta` und `value`. Der Parameter `board` stellt dabei den aktuellen Spiel-Zustand dar, die Werte `alpha` und `beta` sind die Schranken des Alpha-Beta-Prunings. Diese drei Werte werden schlicht übernommen, da die Minimax-Funktion ebenfalls mit diesen Parametern aufgerufen wird. Der letzte Parameter `value` hingegen ist der Rückgabewert der Evaluierungsfunktion `static_eval`.

Bevor die eigentliche Quiescence-Search-Funktion `quiesce` vorgestellt wird, wird in dem Code-Ausschnitt 3.2 die Hilfsfunktion `get_pieces_sum` gezeigt. Diese Funktion hat als Eingabeparameter `board`, den aktuellen Spielzustand, und berechnet lediglich die Figurenanzahl – beziehungsweise die Werte, sodass also eine Dame den Rückgabewert stärker erhöht als ein Bauer – der beiden Spieler. Diese Funktion ist notwendig, um zu überprüfen, ob ein Zug ein taktischer Zug war, also eine Veränderung der Figurenanzahl/-werte verursacht hat.

```
1  def get_pieces_sum(board):
2      value = 0
3      for piece_type in range(1,6):
4          value += len(board.pieces(piece_type,chess.WHITE))
5          value += len(board.pieces(piece_type,chess.BLACK))
6      return value
```

Listing 3.2: Hilfsfunktion für die Differenzierung taktischer Züge

Es existieren zudem zwei Konstanten, `ENABLE_QUIESCENCE_SEARCH` und `QUIESCE_DEPTH`. Bei der ersten Konstante handelt es sich um ein *Flag*, um die Ruhesuche zu deaktivieren. Dies ist für den Test der Funktion notwendig, da überprüft wird, ob das Programm mit Quiescence Search einen besseren Zug spielt, als ohne. Die Konstante `QUIESCE_DEPTH` steht standardmäßig auf 2 und legt fest, wie tief das Programm in der Ruhesuche weiterhin sucht. Eine solche Begrenzung ist notwendig, da das Programm andernfalls womöglich sehr viele Ebenen tiefer rechnen würde, wenn in mehreren aufeinanderfolgenden Zügen eine Figur geschlagen oder aufgewertet werden kann. Wird diese Begrenzung nicht implementiert, endet das Programm in einem Rekursionsfehler.

```
1  def quiesce(board,alpha,beta,value):
2      if not ENABLE_QUIESCENCE_SEARCH:
3      return value
4    pieces_after_move = get_pieces_sum(board)
5    move = board.pop()
6    pieces_before_move = get_pieces_sum(board)
7    board.push(move)
8    if(pieces_after_move != pieces_before_move):
9      value = minimax(board,QUIESCE_DEPTH,max(value,alpha),beta,True)
10   return value
```

Listing 3.3: Funktion für die Implementierung der Ruhesuche

Der Code-Ausschnitt 3.3 zeigt nun die komplette Funktion `quiesce`, welche die Ruhesuche implementiert. Hierfür wird zunächst der bereits evaluierte Wert `value` zurückgegeben, falls die Quiescence Search Erweiterung deaktiviert ist (erkennbar durch das Flag `ENABLE_QUIESCENCE_SEARCH`). Andernfalls wird zunächst die Summe der Figurenwerte aktuell – also nach dem letzten evaluierten Zug –, danach die Summe der Figurenwerte vor jenem letzten Zug verglichen. Stimmen diese beiden Werte nicht überein, so besteht hier die beschrieben Notwendigkeit, die Quiescence Search anzuwenden. In diesem Fall wird die Minimax-Funktion mit den folgenden Parametern aufgerufen:

- `board`: Hier wird der aktuelle Spielzustand `board` übernommen, der der Funktion `quiesce` übergeben wurde. Der Spielzustand ist nicht verändert, da der `move`, welcher zuerst rückgängig gemacht wurde (um die Anzahl/Werte der Figuren zu erfassen), wieder „gespielt“ wurde
- `depth`: Hier wird die Konstante `QUIESCE_DEPTH`, welche oben erwähnt wurde, übergeben. Es wird somit festgelegt, wie weit die Ruhesuche tiefer sucht, falls ein taktischer Zug der letzte analysierte Zug war
- `alpha`: Hier wird der größere Wert von `value` und `alpha` übergeben
- `beta`: Hier wird lediglich der Wert der Variable `beta` übernommen, welcher bei Aufruf der Funktion übergeben wurde
- `is_quiesce`: Hier wird der Wert `True` übergeben. Dadurch wird erreicht, dass, bei der Suchtiefe 0, nicht erneut die Quiescence-Search-Funktion aufgerufen wird

3.4 Testen der Programm-Erweiterung

Um zu überprüfen, ob die Erweiterung, welche den Quiescence Search Algorithmus implementiert, wie erwartet funktioniert, wurde ein weiterer Test geschrieben. Zu diesem Zweck wurde das Szenario aus Abbildung 3.1 übernommen: Das Programm wurde zunächst normal aufgerufen, sodass die Erweiterung für die Quiescence Search in Kraft tritt. Aus Gründen der Einfachheit, wird die Minimax-Funktion mit einer Tiefe von 1 aufgerufen. Der Code-Ausschnitt 3.4 zeigt dies.

```
%%time
board,move = perform_move(chess.Board("b3k3/1n5p/6p1/8/8/8/7P/1Q4K1 w
    - - 0 1"),1, True)
assert str(move) != "b1b7"
```

Listing 3.4: Test für die Quiescence Search (normaler Aufruf)

In dem gezeigten Code-Ausschnitt wird die Funktion `perform_move` verwendet, welche jedoch lediglich das mehrfache Schreiben gleicher Anweisungen verhindert. Es wird geprüft, ob es sich bei dem errechneten Zug **nicht** um den Zug *b1b7*, also die Dame auf B7, wo sie im nächsten Zug geschlagen werden kann, handelt. Abbildung 3.2 zeigt, dass dies der Fall ist und die Ruhesuchenerweiterung somit eingreift und auch den taktischen Zug erkannt hat.

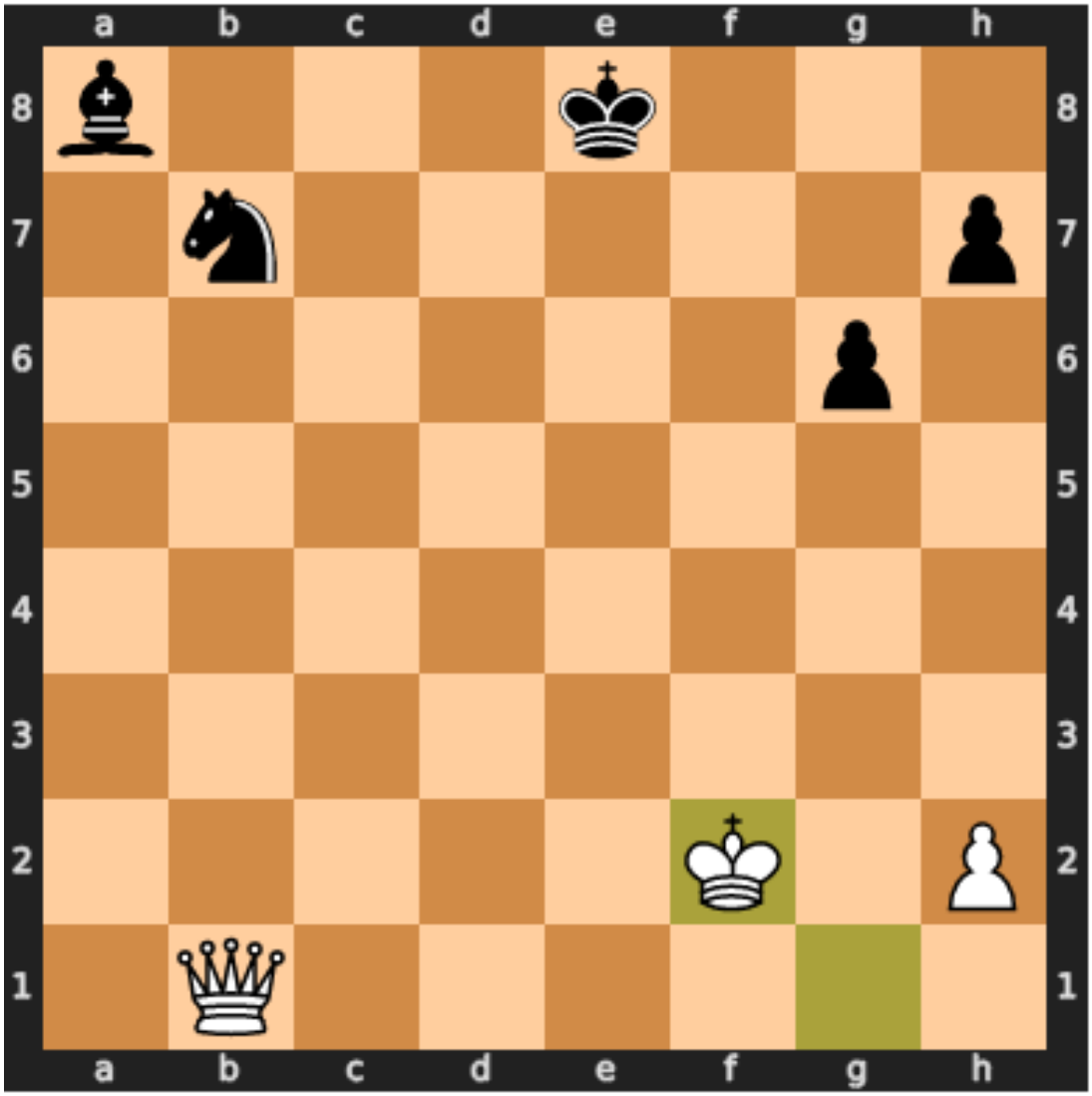

Abbildung 3.2: Korrekte Anwendung der Quiescence Search: Der scheinbar gute Zug wird nicht gewählt, obwohl die Minimax-Tiefe auf 1 gesetzt ist

Zur Kontrolle des Tests wird der Test erneut aufgerufen, jedoch wird bei dem zweiten Mal das Flag `ENABLE_QUIESCENCE_SEARCH` auf `False` gesetzt, sodass die Erweiterung der Ruhesuche nicht verwendet wird (Der Code-Ausschnitt 3.1 zeigt dies). Der Code-Ausschnitt 3.5 zeigt der Vollständigkeit halber auch den Code für diesen zweiten Test.

```
%%time
ENABLE_QUIESCENCE_SEARCH = False
CACHE = {} # Has to be cleared for this test to work
board,move = perform_move(chess.Board("b3k3/1n5p/6p1/8/8/8/7P/1Q4K1 w
    - - 0 1"),1, True)
assert str(move) == "b1b7"
ENABLE_QUIESCENCE_SEARCH = True
```

Listing 3.5: Test für die Quiescence Search (Aufruf mit deaktivierter Ruhesuche)

Es ist zu sehen, dass der Test nahezu gleich ist, jedoch einen anderen errechneten Zug erwarten, nämlich den scheinbar guten Zug *b1b7*. Abbildung 3.3 zeigt, dass das Ergebnis dieses Tests auch das eben erwartete ist: Das Programm denkt, aufgrund der Suchtiefe, dass es den besten Zug gefunden hat; der Horizonteffekt tritt ein.

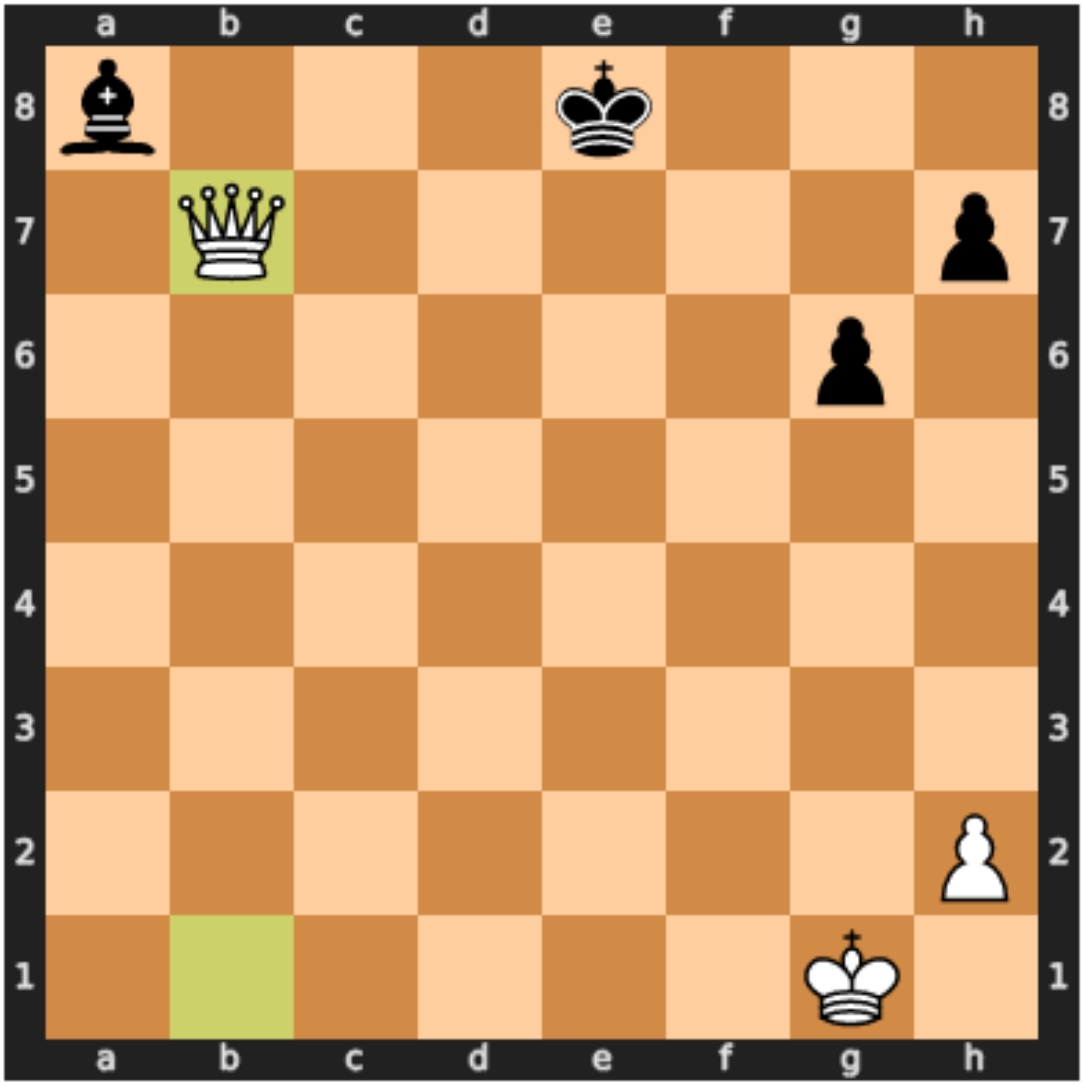

Abbildung 3.3: Kontrolltest: Ohne Quiescence Search wählt das Programm einen schlechten Zug, welchen es allerdings aufgrund des Horizonteffekts nicht als solchen erkennt

3.5 Performanz-Analyse

Letztlich gilt es zu untersuchen, inwiefern die Verbesserung der Zugberechnung Auswirkungen auf die Performanz und Berechnungsdauer des Programms hat. Hierfür wurde eine Performanz-Analyse gemacht, in der die Tests für die Schach-KI mehrfach hintereinander ausg eführt werden. Insgesamt wurden die Tests mehrfach ausgeführt, hierbei jeweils fünfmal für die folgenden Szenarios:

- Suchtiefe 4, keine Verwendung von Quiescence Search (Szenario **A**)
- Suchtiefe 4, Verwendung von Quiescence Search mit Tiefe 1 (Szenario **B**)
- Suchtiefe 4, Verwendung von Quiescence Search mit Tiefe 2 (Szenario **C**)

- Suchtiefe 5, keine Verwendung von Quiescence Search (Szenario **D**)
- Suchtiefe 5, Verwendung von Quiescence Search mit Tiefe 1 (Szenario **E**)
- Suchtiefe 5, Verwendung von Quiescence Search mit Tiefe 2 (Szenario **F**)

Tabelle 3.1 stellt die Ergebnisse dar. Die Buchstaben in der obersten Reihe korrespondieren dabei zu den oben erwähnten Szenarios. Die Werte in der zweiten Zeile beschreiben die durchschnittliche Dauer, bis alle Tests durchgelaufen sind. Sie lassen keine Rückschlüsse darauf zurück, wie lange die Berechnung eines einzelnen Zuges dauert. Letztlich ist es immer im Ermessen des Spielers, wie lange die Zugberechnung dauern darf, bis das Programm „unspielbar“ wird.

A	**B**	**C**	**D**	**E**	**F**
Ø 871,05 s	Ø 868,81 s	Ø 872,07 s	Ø 4.423,37 s	Ø 4.464,97 s	Ø 4.645,28 s

Tabelle 3.1: Ergebnisse der Performanz-Analyse

Die Ergebnisse zeigen, dass die Performanz-Unterschiede bei allen Szenarios mit einer Suchtiefe von 4 (**A**, **B** und **C**) sehr marginal sind und sich wahrscheinlich auf Unterschiede bei den Ausführungsbedingungen – sprich aufgrund der Hardware – zurückführen lassen; schließlich ist die durchschnittliche Dauer für das Szenario **B** geringer als jene für das Szenario **A**, obwohl im Ersteren mehr Berechnungen ausgeführt werden. Außerdem zeigt die Tabelle, dass die Erhöhung der Suchtiefe auf 5 die Berechnungsdauer stark erhöht; auch hier ist der Unterschied zwischen der Verwendung keiner Quiescence Search und der Verwendung dieser mit einer weiteren Suchtiefe von 1 ebenfalls sehr gering. Lediglich bei der Verwendung der Quiescence Search mit einer Tiefe von 2 ist ein stärkerer Unterschied zu betrachten. Schlussfolgernd lässt sich festhalten, dass die Verwendung einer Ruhesuche die Performanz nicht schwerwiegend verschlechtert. Betrachtet man zudem, dass die Verwendung einer Quiescence Search die Zugqualität steigert, spricht vieles für die Implementierung dieses Verfahrens.

4 Fazit und Ausblick

Zusammenfassend lässt sich sagen, dass Programme wie die entwickelte Schach-KI stets viele Ansatzpunkte für Verbesserungen haben. Die Verbesserung in Form der Quiescence Search wurde in dieser Arbeit analysiert. Auch, wenn die Erweiterung zur Folge hat, dass die Berechnung der einzelnen Züge länger dauert, handelt es sich bei dem Horizonteffekt um ein Problem mit starken Auswirkungen, weshalb die Implementierung einer Quiescence Search von sehr großem Vorteil für die generelle Qualität der errechneten Züge ist.

Als mögliche Weiterentwicklung der Implementierung bietet sich an, eine größere Menge an taktischen Zügen zu definieren. Schließlich kann der Horizonteffekt potentiell bei jedem Zug auftreten; auch hier ist es eine Abwägung von Performanz-Einbußungen und Zug-Verbesserungen. Außerdem könnte man die Suchtiefe der Quiescence Search so umformen, dass das Programm so lange sucht, bis ein de facto „ruhiger“ Zug gefunden wurde, statt eine festgelegte Tiefe von beispielsweise 2 oder 3 zu verwenden.

Generell war die Implementierung der Ruhesuche erfolgreich und hat die Gesamtqualität der Schach-KI verbessert. Im Vergleich zum gesamten Programm des Minimax-Algorithmus handelt es sich bei der Ruhesuchenerweiterung um einen eher kleinen Teil, welcher jedoch große Auswirkungen hat. Dies spricht zweifellos für die Verwendung einer Quiescence Search bei derartigen Programmen.

Literatur

Chess Programming Wiki (2021). *Tactical Moves - Chessprogramming wiki*. URL: https://www.chessprogramming.org/Tactical_Moves (besucht am 07. 07. 2021).

Merz, Peter (2002). *Moderne heuristische Optimierungsverfahren: Meta-Heuristiken*. URL: http://www.ra.cs.uni-tuebingen.de/lehre/ss02/vs_mh/mh-v1.pdf (besucht am 06. 07. 2021).

Oxford Reference (2021). *Horizon effect - Oxford Reference*. URL: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095944934 (besucht am 07. 07. 2021).

Shannon, Claude E. (1950). „XXII. Programming a Computer for Playing Chess“. In: *Philosophical Magazine* 41.314.

Wikipedia (2021a). *Grafik-Upload Alpha-Beta-Pruning*. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/9/91/AB_pruning.svg/1200px-AB_pruning.svg.png (besucht am 06. 07. 2021).

– (2021b). *Grafik-Upload Minimax-Algorithmus*. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Minimax.svg/400px-Minimax.svg.png (besucht am 06. 07. 2021).