Erstellung einer Schach KI

Studienarbeit

des Studiengangs Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Florian Dahlitz

und

Moritz Heinz

Abgabedatum: 29.04.2020

Bearbeitungszeitraum	14.09.19-29.04.20
Matrikelnummer, Kurs	1867805, TINF17AIBC
Matrikelnummer, Kurs	9070408, TINF17AIBC
Betreuer der Hochschule	Prof. Dr. Karl Stroetmann

Abstract in deutscher Sprache

Die vorliegende Arbeit stellt die Implementierung eines Schachcomputers dar. Hierfür werden zuerst die theoretischen Grundlagen, die für die Erstellung des Schachcomputers notwendig sind, erläutert. Dies umfasst den Mini-Max-Algorithmus, die Bewertungsfunktion und die in der Arbeit verwendete Bewertungsheuristik sowie die Alpha-Beta-Suche. Des Weiteren werden als Verbesserungen der Alpha-Beta-Suche die Zugvorsortierung, die Ruhesuche und Transpositionstabellen erklärt. Es folgt die Implementierung des Schachcomputer mittels der Programmiersprache Python. Den Abschluss der Arbeit bildet ein Auswertungs- und Diskussionsteil.

Abstract in englischer Sprache

The paper at hand shows the implementation of a chess computer. Therefore, theoretical basics such as the Mini-Max-Algorithm, the used evaluation heuristic as well as Alpha-Beta-Pruning are explained. Furthermore, enhancements of Alpha-Beta-Pruning namely move ordering, quiescence search, and transpositional tables are described. Subsequently, the actual implementation of the chess computer is shown followed by an evaluation and discussion section.

Inhaltsverzeichnis

Abkürzungsverzeichnis

B Bauer

D Dame

FEN Forsyth-Edwards Notation

K König

L Läufer

LRU Least-Recently-Used

PyPI Python Package Index

S Springer

T Turm

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

„Es lässt sich nicht mit Gewissheit belegen, wo und wann die Menschen zum ersten Mal Schach gespielt haben. Die meisten Quellen deuten darauf hin, dass das heute beliebteste Strategiespiel der Welt im 6. Jahrhundert in Indien populär wurde, über Persien in die arabischen Länder gelangte und von dort aus seinen Siegeszug um die ganze Welt angetreten hat.“[Schachverein Gifhorn, o. J.]

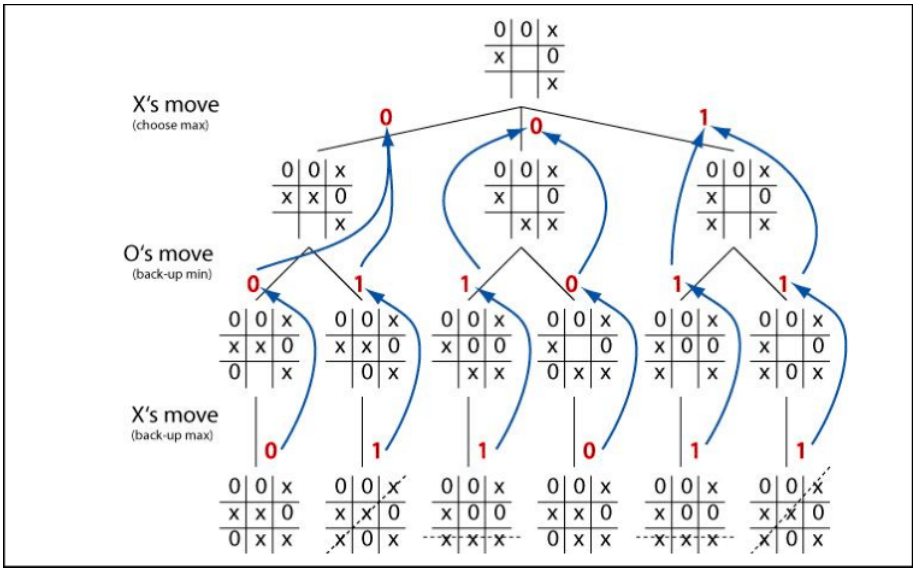
Diese ursprünglichen Formen des Schachs waren dem heutigen Spiel sehr fern. Im 13. Jahrhundert wurden die Spielregeln angepasst, um das Spiel dynamischer und schneller zu machen. Mit der im 16. Jahrhundert hinzugekommenen Rochade umfasste das Schachspiel die Regeln, die es bis heute mit einigen Ausnahmen hat [Schachverein Gifhorn, o. J.]. Auch heute noch ist Schach ein sehr beliebtes Spiel [Schach-Club Kreuzberg, o. J.], für das mittlerweile auch viele Online Varianten existieren. Dabei kann man gegen reale Gegner oder gegen Schachcomputer spielen. Im Rahmen des Studiums und vor dem Hintergrund verschiedener Vorlesungen mit dem Bezug zur künstlichen Intelligenz wird im Zuge dieser Studienarbeit ein Schachcomputer entwickelt.

Hierzu werden zuerst einige theoretische Grundlagen, die für die Erstellung des Schachcomputers notwendig sind, erläutert. Dies umfasst die Erarbeitung des Mini-Max-Algorithmus, die verwendete Bewertungsheuristik sowie die Alpha-Beta-Suche. Ferner werden bei der Alpha-Beta-Suche die Vorsortierung, die Ruhesuche und Transpositionstabellen als Verbesserungen eingeführt. Es folgt die Implementierung des Schachcomputers, die in einem *Jupyter Notebook* erfolgt. Für die Visualisierung und den Spielablauf wird zudem die Python Bibliothek *Python-Chess* verwendet. Es folgt eine Auswertung der Implementierung und eine Diskussion über vernachlässigte Aspekte in dieser Arbeit.

2 Theoretische Grundlagen

2.1 Mini-Max-Suche

Im Schachspiel ist jeder Spieler bestrebt, das für ihn bestmögliche Spielergebnis zu erzielen. Zur Abbildung dieses Prinzips dient der *Mini-Max*-Algorithmus [Russell und Norvig, 2010]. Um dies abbilden zu können wird ein Punktwert eingeführt, den ein Spieler zu maximieren versucht und der andere Spiele zu minimieren versucht. Für das Schachspiel und die spätere Betrachtung dessen wird an dieser Stelle festgelegt, dass eine hohe Bewertungen für einen Vorteil der weißen Spielseite steht, wohingegen eine negative Bewertungen für einen Vorteil für die schwarze Spielseite steht. Somit ist Weiß bestrebt, den Wert durch die Wahl einer Spieloption zu maximieren und Schwarz den Wert zu minimieren [Paulsen, 2009]. Zur Berechnung dieses Punktwertes wird ein Spielbaum (engl. *game tree*) benötigt, der alle möglichen Spielzüge beinhaltet. Es werden alle Knoten eines Spielbaumes mit ihren zugehörigen Werten generiert [Shah, 2007]. Die Knoten innerhalb des Baumes werden in drei verschiedene Kategorien unterteilt: Blattknoten, minierende und maximierende Knoten. Jeder Blattknoten erhält seinen Nutzwert (engl. *utility value*). Den minimierenden Knoten wird jeweils der kleinste Wert ihrer Kindknoten zugewiesen, den maximierenden Knoten jeweils der größte Wert [Shah, 2007]. Abbildung 1 zeigt einen Teil eines Spielbaums für das Spiel Tic-Tac-Toe, der die einzelnen Schritte der Berechnung durch den Mini-Max-Algorithmus beinhaltet.



Die Ausgangssituation des Spiel ist an der Wurzel des Baumes abgebildet. Im Falle des Spiels Tic-Tac-Toe wird festgelegt, dass Spieler X bestrebt ist, den Punktwert zu maximieren, wohingegen Spieler O bestrebt ist, diesen zu minimieren. Spieler X ist als nächstes am Zug und besitzt drei Möglichkeiten, sein Kreuz auf dem Spielfeld zu platzieren. Dem folgt der Spielzug des Spielers O, der seinerseits für jeden möglichen Spielzug von Spieler X je zwei Möglichkeiten besitzt, sein Symbol zu setzen. Somit gibt es von der Ausgangslage her sechs mögliche Spielzüge, die Spieler O durchführen kann. Zu guter Letzt hat Spieler X für jede seiner Ausgangslagen nur noch eine Möglichkeit, sein Kreuz zu setzen. Jeder mögliche Endstand des Spiels erhält nun einen Punktwert. Markiert der Endstand einen Sieg für Spieler X, so erhält dieser den Wert 1, bei einem Sieg für Spieler O den Wert -1 und bei einem Unendschieden den Wert 0. Die Werte der Blattknoten werden anschließend nach oben propagiert. Maximierende Knoten (Spieler X ist am Zug) übernehmen den jeweils höchsten Zahlenwert, minimierende Knoten (Spieler O ist am Zug) den niedrigsten Wert. Am Ende dieses Prozesses ist von der Ausgangslage des Spiels erkennbar, dass Spieler X dem rechten Pfad des Baumes folgen muss, um garantiert einen Sieg zu erzielen, es für ihn aber auch bei einer Fehlentscheidung im nächsten Zug noch möglich ist, das

Spiel zu gewinnen. Beim Schach ist es jedoch in der Regel nicht möglich alle Positionen bis zum Spielende zu analysieren. Im Gegenteil zu Tic-Tac-Toe, bei dem der Spielbaum maximal die Tiefe neun erreicht, kann beim Schach theoretisch eine Tiefe von 5899 erreicht werden [Wikipedia, 2018]. Da es im Verlauf eines Schachspiels durchschnittlich x mögliche Züge für den Spieler am Zug gibt, gibt es ab einer bestimmten Tiefe t, x^t zu analysierenden Positionen. Weil sogar schnelle Computer bei diesem exponentiell wachsenden Baum zu hohe Rechenzeiten benötigen, muss eine maximale Tiefe, bis zu der der Spielbaum analysiert wird, festgelegt werden. Diese ist je nach Anspruch bei präzisen Evaluationen tiefer, bei schnellen Evaluationen weniger tief. Um den Minimax Algorithmus auf Programmierebene umzusetzen, werden folgende Methoden benötigt [Shannon, 1950]:

- Eine Methode, welche für eine Stellung die erlaubten Züge bestimmen kann. Hierfür müssen die verschiedenen Zugmöglichkeiten der Figuren, Schach und Sonderzüge wie En-Passant und die Rochade berücksichtigt werden.
- Eine Methode, die die Kernaufgabe des Minimax Algorithmus umsetzt, also das abwechselnde Maximieren und Minimieren der möglichen Folgezüge einer Stellung.
- Eine Methode, die die Bewertungsheuristik umsetzt.
- Eine Methode, welche für eine Position Schachzüge ausführen und diese auch rückgängig machen kann.

Bei der Umsetzung der Kernaufgabe des Minimax-Algorithmus wird der Ansatz von [Knuth und Moore, 1975] übernommen. Außerdem wird die Standard-Implementierung verwendet, bei der der gesamte Algorithmus in zwei Prozeduren, Minimize und Maximize, aufgeteilt wird. Diese führen dabei jene Tätigkeiten aus, welche die Entscheidungen des minimierenden bzw. maximierenden Spielers repräsentieren. Bei der alternativen Negamax-Implementierung gibt es nur eine Minimax-Prozedur

welche die Funktionen beider Spieler übernimmt [Wikipedia, 2020]. Ist ein Spielbaum gegeben, so lässt sich die optimale Strategie basierend auf den Minimax-Werten (engl. *minimax values*) der einzelnen Knoten ableiten. Diese optimale Strategie wird als MINIMAX(n) geschrieben und ist in Formel 1 dargestellt [Russell und Norvig, 2010].

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & * \\ max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & ** \\ min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & *** \end{cases} \quad (1)$$

* if TERMINAL-TEST(s)

** if PLAYER(s) = MAX

*** if PLAYER(s) = MIN

Dabei gelten folgende Aussagen [Russell und Norvig, 2010]:

- s ist der aktuelle Zustand bzw. Knoten im Spielbaum.
- a ist eine Aktion bzw. ein Halbzug.
- Die Funktion UTILITY gibt den Nutzwert für einen Blattknoten zurück.
- Die Funktion PLAYER gibt den Spieler zurück, der am Zug ist (MAX für Weiß und MIN für Schwarz).
- Die Funktion Actions liefert die Menge der für den übergebenen Zustand gültigen Züge.
- TERMINAL-TEST gibt den Wert *true* zurück, wenn das Spiel zu Ende ist (erreichen eines Blattknotens), ansonsten $false$.
- RESULT liefert den Zustand zurück, der folgt, wenn auf den Zustand s die Aktion a angewandt wird.

Für den Minimax-Wert, den die Funktion MINIMAX liefert, wird angenommen, dass beide Spieler bis zum Ende des Spiels optimal spielen [Russell und Norvig, 2010]. Da der Minimax-Algorithmus in dieser Form

sehr rechenintensiv ist und viel Speicher benötigt, wird er durch eine maximale Tiefe begrenzt (*begrenzte Tiefensuche*). Zur Umsetzung dieser begrenzten Tiefensuche werden Funktionen F (Formel 2) für den Spieler Weiß und G (Formel 3) für Spieler Schwarz definiert, dafür wird er Ansatz von [Knuth und Moore, 1975] verwendet. Die beiden Funktionen definieren das Verhalten der Spieler beim Treffen auf eine Schachposition p in der Tiefe t.

$$F(p,t) = \begin{cases} e(p) \quad \text{if } t = 0, \\ max(G(p_1, t-1), G(p_2, t-1), ..., G(p_l, t-1)) \quad \text{if } t > 0 \end{cases} \tag{2}$$

$$G(p,t) = \begin{cases} e(p) \quad \text{if } t = 0, \\ min(F(p_1, t-1), F(p_2, t-1), ..., F(p_l, t-1)) \quad \text{if } t > 0 \end{cases} \tag{3}$$

Dabei sind p_1, ... , p_l die möglichen Positionen, welche nach einem Zug aus der Position p resultieren können und $e(p)$ die Bewertungsfunktion ist. Der Pseudocode für einen Minimax mit Tiefenbegrenzung ist in Algorithmus 1 dargestellt. Die verwendeten Hilfsfunktionen $p.move(m)$ und $p.undoMove(m)$ führen auf der Schachstellung p den Zug m aus, beziehungsweise machen ihn rückgängig. Es kann, anders als bei der Negamax-Implementierung, dieselbe Bewertungsfunktion für beide Funktionen benutzt werden, weil diese weiß-favorisierende Positionen mit positiven Werten und schwarz-favorisierende Positionen mit negativen Werten bewertet. Die Zahl d repräsentiert die Tiefe (engl. *depth*), $evaluate$ ist die Bewertungsfunktion und die Hilfsfunktion $findLegalMoves(p)$ gibt die Menge der zulässigen Halbzüge für p zurück. Der Punktwert (engl. *score*) wird in der Variablen $score$ gespeichert.

Algorithmus 1 Minimax mit Tiefenbegrenzung

```
1:  function MAXIMIZE(Integer d, Position p)
2:      if d < 1 then
3:          return evaluate(p)
4:      else
5:          score ← −∞
6:          legalMoves ← findLegalMoves(p)
7:          for all m in legalMoves do
8:              p.move(m)
9:              moveScore ← minimize(d − 1, p)
10:             p.undoMove(m)
11:             if moveScore > score then
12:                 score ← moveScore
13:             end if
14:         end for
15:     end if
16:     return score
17: end function
18: function MINIMIZE(Integer d, Position p)
19:     if d < 1 then
20:         return evaluate(p)
21:     else
22:         score ← ∞
23:         legalMoves ← findLegalMoves(p)
24:         for all m in legalMoves do
25:             p.move(m)
26:             moveScore ← maximize(d − 1, p)
27:             p.undoMove(m)
28:             if moveScore < score then
29:                 score ← moveScore
30:             end if
31:         end for
32:     end if
33:     return score
34: end function
```

2.2 Bewertungsfunktion und -heuristik

In der Spieltheorie im Allgemeinen und im Schach im Besonderen ist es in der Regel nicht möglich, alle möglichen Zugfolgen aus einer Spielposition heraus bis zum Ende zu verfolgen. Deshalb „[...] wird eine Funktion benötigt, die die Stellung auf dem Spielbrett danach bewertet, ob sie für eine der beiden Parteien vorteilhaft oder nachteilig ist.“ [Paulsen, 2009] Diese Funktion wird als *Bewertungsfunktion* bezeichnet. Die Bewertungsfunktion setzt sich aus einem materiellen und einer positionellen Komponente zusammen. Bei der materiellen Komponente werden zunächst die vorhandenen weißen Figuren nach Stärke bewertet und aufsummiert. Anschließend werden die schwarzen Figuren nach Stärke bewertet, aufsummiert und vom Gesamtwert der weißen Figuren subtrahiert. Daraus lassen sich erste Schlussfolgerungen über den Spielstand ziehen. Zudem ist es möglich, die derzeitige Spielphase abzuleiten [Paulsen, 2009].

Da es beim Schach aber auch entscheidend ist, auf welchen Feldern sich die einzelnen Figuren befinden und in welcher Position sie zueinander stehen (Bauernstruktur, Königssicherheit), wird zusätzlich zur materiellen Komponente eine positionelle Komponente berechnet. Die verschiedenen Positionen werden aus der Spielbrettaufstellung entnommen und fließen mit ihren jeweiligen Gewichtungen in die anschließende Bewertung ein [Paulsen, 2009].

2.2.1 Einfache Bewertungsheuristik

Das Schachspiel setzt sich aus zwei wichtigen Faktoren zusammen: Einerseits ist abgesehen von der Tatsache, welcher Spieler den ersten Zug macht, keine Zufallskomponete enthalten, andererseits handelt es sich um ein Spiel mit perfekter Information. Diese zwei Faktoren führen dazu, dass bei jeder Stellung eine der folgenden drei Aussagen gilt [Shannon, 1950]:

1. Es handelt sich um eine gewonnene Position für Weiß. Somit kann Weiß einen Sieg forcieren, wobei Schwarz verteidigt.
2. Es handelt sich um eine gewonnene Position für Schwarz. Somit kann Schwarz einen Sieg forcieren, wobei Weiß spielt.
3. Es handelt sich um eine ausgeglichene Position für beide Parteien. Somit kann es nur ein Unentschieden am Ende geben, falls beide Parteien keine Fehler machen.

Bei einigen Spielen (so auch beim Schach) lässt sich aus den genannten zwei Faktoren und den daraus resultierenden drei Aussagen eine *Bewertungsfunktion* $f(P)$ ableiten, wobei P die Stellung bezeichnet. Der Rückgabewert der Funktion ist die Kategorie, in die die jeweilige Position gehört: Sieg (+1), Unentschieden (0), Niederlage (-1). Zum Zeitpunkt des Zuges des Schachcomputers werden die Werte $f(P)$ für alle Positionen nach möglichen Halbzügen berechnet. Der Zug mit dem maximalen Wert wird am Ende ausgeführt [Shannon, 1950].

Es ist nicht realisierbar, alle Spielzugmöglichkeiten durchzurechnen. Aus diesem Grund werden Approximationen der Bewertungsfunktion entworfen. Diese Approximationen werden als Heuristiken bzw. als Bewertungsheuristiken bezeichnet. Im Zuge dieser Arbeit wird die einfache Bewertungsheuristik nach Tomasz Michniewski verwendet, die ursprünglich in der *Polish chess programming discussion list (progszach)* veröffentlicht wurde und im chessprogramming.org Wiki beschrieben wird. Die Bewertungsheuristik ist bewusst einfach gehalten, da sie aufgrund ihrer Einfachheit schneller ist [Wiki, 2018]. Die von Tomasz Michniewski beschriebene einfache Bewertungsheuristik wird in zwei Teilen dargestellt: Figurenwerte (engl. *simple piece values*) und figurenspezifische Positionstabellen (engl. *piece-square tables*).

Mit der Festlegung von Werten je Figur werden vier verschiedene Ziele erreicht:

1. Vermeidung des Austauschs einer leichten Figur gegen drei Bauern

2. Dem Computer signalisieren, dass das Halten des Läuferpaars vorteilhaft ist
3. Vermeidung des Austauschs von zwei leichten Figuren gegen einen Turm und einen Bauern
4. Ein Turm und zwei Bauern reichen aus, um gegen zwei leichte Figuren zu gewinnen

Der erste Punkt wird durch Formel 4 erfüllt.

$$\begin{aligned} L &> 3B \\ S &> 3B \end{aligned} \tag{4}$$

Zwar gibt es durchaus Positionen, in denen drei Bauern wertvoller als eine leichte Figur sind, jedoch ist es im Allgemeinen besser eine leichte Figur zu behalten, da im Spielverlauf die Bauern individuell attackiert werden können und deren Wert als Dreier-Figuren-Gespann verloren geht [Wiki, 2018]. Der zweite genannte Punkt wird durch Formel 5 erreicht.

$$L > S \tag{5}$$

Zwar garantiert diese Formel kein Halten des Läuferpaars, da am Ende ein Läufer gegen einen Springer stehen kann, dennoch ist es eine Tatsache, dass Spieler oftmals Springer mit Läufern schlagen und nicht Läufer mit Springern [Wiki, 2018]. Die ersten beiden Formeln 4 und 5 führen zusammen zu Formel 6.

$$L > S > 3B \tag{6}$$

Der dritte Punkt wird zwar durch Formel 7 erreicht, dennoch haben Spiele wie jenes zwischen Karpov und Kasparov gezeigt, dass bereits ein Turm und zwei Bauern ausreichen, um gegen zwei leichte Figuren gewinnen zu können.

$$L + S > T + B \tag{7}$$

Aus diesem Grund wird die Formel 7, die zu Formel 8 erweitert werden kann, um einen Faktor erweitert, woraus sich Formel 9 ergibt [Wiki, 2018].

$$T + 2B > L + S > T + B \tag{8}$$

$$L + S > T + 1.5B \tag{9}$$

Durch die hier beschriebenen Formeln ist somit auch der vierte Punkt erfüllt. Zu guter Letzt wird noch eine Formel benötigt, die das Verhältnis einer Dame-Bauer-Kombination gegenüber zwei Türmen darstellt. Dies ist in Formel 10 abgebildet.

$$D + B = 2T \tag{10}$$

Somit erhält man das Gleichungssystem, das in Formel 11 dargestellt ist.

$$\begin{aligned} L > S > 3B \\ L + S = T + 1.5B \\ D + B = 2T \end{aligned} \tag{11}$$

Dieses Gleichungssystem wird durch die in Tabelle 1 gelisteten Werte erfüllt. Diese Werte wurden von Tomasz Michniewski festgelegt mit Ausnahme des Wertes des Königs. Dieser Wert stammt aus dem Paper von Claude Shannon, wobei dieser an der Stelle dem König den Wert 200 gibt, was durch die Umrechnung in Hundertstelbauer zu 20000 führt [Shannon, 1950]. Der Wert für den König ist bewusst so hoch gewählt worden, da der Verlust des Königs automatisch zur Niederlage führt. Daraus folgt zudem, dass der Verlust des Königs durch die materielle Komponente der Bewertungsheuristik einfacher erkannt werden kann. Zu guter Letzt sei angemerkt, dass die Wahl der Figurenwerte dazu führt, dass diese in einer 2 Byte vorzeichenbehafteten Ganzzahl abgelegt werden können, da

Figur	**Wert**
Bauer	100
Springer	320
Läufer	330
Turm	500
Dame	900
König	20000

der Gesamtwert aller Spielfiguren einer Seite rund 30300 beträgt [Wiki, 2018].

Nachdem die Figurenwerte ausführlich dargelegt wurden, werden nun figurenspezifische Positionstabellen erstellt. Diese haben die Aufgabe für gut positionierte Figuren eine *Belohnung* und für schlecht positionierte Figuren *Abzüge* zu erteilen [Wiki, 2018].

Für die Bauern gilt, dass deren Vorwärtsbewegung prinzipiell belohnt wird. Des Weiteren werden die zentralen Bauern (D2, E2) negativ bewertet, sollten sie sich nicht bewegen. Das rührt daher, da Bauern direkt vor dem König-Dame-Paar als hindernd angesehen werden, weil es im Besonderen den Bewegungsfreiraum der beiden Läufer zu Beginn des Spiels einschränkt.[Wiki, 2018]. Formel 12 zeigt mittels einer Matrix die positionsbezogenen Werte für Bauern auf dem Spielbrett.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 50 & 50 & 50 & 50 & 50 & 50 & 50 & 50 \\ 10 & 10 & 20 & 30 & 30 & 20 & 10 & 10 \\ 5 & 5 & 10 & 25 & 25 & 10 & 5 & 5 \\ 0 & 0 & 0 & 20 & 20 & 0 & 0 & 0 \\ 5 & -5 & -10 & 0 & 0 & -10 & -5 & 5 \\ 5 & 10 & 10 & -20 & -20 & 10 & 10 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{12}$$

Eine genauere Erläuterung zu den einzelnen Werten pro Spielposition der Bauern kann bei [Wiki, 2018] nachgelesen werden.

Springer sind aufgrund ihrer Bewegungsart am effektivsten in der Mitte des Spielfeldes, weshalb sie auf den mittleren Positionen höhere Punktwerte erhalten. Demgegenüber erhalten sie negative Punktwerte für Randpositionen, da diese ihren Bewegungsradius einschränken [Wiki, 2018]. Formel 13 zeigt die entsprechenden Werte für die Springerpositionen.

$$\begin{pmatrix} -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \\ -40 & -20 & 0 & 0 & 0 & 0 & -20 & -40 \\ -30 & 0 & 10 & 15 & 15 & 10 & 0 & -30 \\ -30 & 5 & 15 & 20 & 20 & 15 & 5 & -30 \\ -30 & 0 & 15 & 20 & 20 & 15 & 0 & -30 \\ -30 & 5 & 10 & 15 & 15 & 10 & 5 & -30 \\ -40 & -20 & 0 & 5 & 5 & 0 & -20 & -40 \\ -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \end{pmatrix} \tag{13}$$

Läufer erhalten zusätzliche Punkte für zentrale Positionen, da sie dort, ähnlich wie Springer, den größten Bewegungsradius besitzen. Des Weiteren erhalten sie für Randpositionen Abzüge, wobei die Positionen in den Ecken des Spielbretts besonders negativ bewertet sind [Wiki, 2018].

Eine positionsbezogene Punktwerteaufstellung mittels einer Matrix ist in Formel 14 abgebildet.

$$\begin{pmatrix} -20 & -10 & -10 & -10 & -10 & -10 & -10 & -20 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & 0 & 5 & 10 & 10 & 5 & 0 & -10 \\ -10 & 5 & 5 & 10 & 10 & 5 & 5 & -10 \\ -10 & 0 & 10 & 10 & 10 & 10 & 0 & -10 \\ -10 & 10 & 10 & 10 & 10 & 10 & 10 & -10 \\ -10 & 5 & 0 & 0 & 0 & 0 & 5 & -10 \\ -20 & -10 & -10 & -10 & -10 & -10 & -10 & -20 \end{pmatrix} \tag{14}$$

Tomasz Michniewski weißt den Türmen keine stark unterschiedlichen positionsbezogenen Punktwerte zu, wie er dies bei den Springern und Läufern tut. Die Türme erhalten lediglich einen Bonus für den siebten Rang und negative Werte für die Außenpositionen A und H [Wiki, 2018] (siehe Formel 15).

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 10 & 10 & 10 & 10 & 10 & 10 & 5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ 0 & 0 & 0 & 5 & 5 & 0 & 0 & 0 \end{pmatrix} \tag{15}$$

Die Erstellung einer Figuren-Quadrat Tabelle für die Dame dient nicht dazu, gute Stellungen zu belohnen, sondern schlechte mit Abzügen zu versehen. Dennoch lässt sich aus der von Tomasz Michniewski aufge-

stellten Darstellung ableiten, dass die Dame in der Mitte des Spielbretts aufhalten sollte [Wiki, 2018] (siehe Formel 16).

$$\begin{pmatrix} -20 & -10 & -10 & -5 & -5 & -10 & -10 & -20 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & 0 & 5 & 5 & 5 & 5 & 0 & -10 \\ -5 & 0 & 5 & 5 & 5 & 5 & 0 & -5 \\ 0 & 0 & 5 & 5 & 5 & 5 & 0 & -5 \\ -10 & 5 & 5 & 5 & 5 & 5 & 0 & -10 \\ -10 & 0 & 5 & 0 & 0 & 0 & 0 & -10 \\ -20 & -10 & -10 & -5 & -5 & -10 & -10 & -20 \end{pmatrix} \tag{16}$$

Zu guter Letzt werden zwei figurenspezifische Positionstabellen für den König aufgestellt. Die erste Tabelle (siehe Formel 17) ist für mittlere Phase des Spiels gedacht. Hierbei wird deutlich, dass der König zurückgehalten und hinter der Bauernlinie gehalten werden sollte [Wiki, 2018].

$$\begin{pmatrix} -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -20 & -30 & -30 & -40 & -40 & -30 & -30 & -20 \\ -10 & -20 & -20 & -20 & -20 & -20 & -20 & -10 \\ 20 & 20 & 0 & 0 & 0 & 0 & 20 & 20 \\ 20 & 30 & 10 & 0 & 0 & 10 & 30 & 20 \end{pmatrix} \tag{17}$$

Für die Endspielphase sieht die Tabelle für den König jedoch anders aus (siehe Formel 18). Randpositionen schränken den König in seiner Bewegung zusätzlich ein und sind gerade in der Endphase von erhöhtem Risiko, weshalb diese Positionen besonders negativ bewertet sind.

Demgegenüber sind die zentralen Positionen besonders positiv bewertet [Wiki, 2018].

$$\begin{pmatrix} -50 & -40 & -30 & -20 & -20 & -30 & -40 & -50 \\ -30 & -20 & -10 & 0 & 0 & -10 & -20 & -30 \\ -30 & -10 & 20 & 30 & 30 & 20 & -10 & -30 \\ -30 & -10 & 30 & 40 & 40 & 30 & -10 & -30 \\ -30 & -10 & 30 & 40 & 40 & 30 & -10 & -30 \\ -30 & -10 & 20 & 30 & 30 & 20 & -10 & -30 \\ -30 & -30 & 0 & 0 & 0 & 0 & -30 & -30 \\ -50 & -30 & -30 & -30 & -30 & -30 & -30 & -50 \end{pmatrix} \tag{18}$$

Zusätzlich zu den eben beschriebenen figurenspezifischen Positionstabellen beschreibt Tomasz Michniewski, wann die Endphase des Spiel beginnt. Diese beginnt nach ihm, wenn eine der beiden folgenden Aussagen zutrifft [Wiki, 2018]:

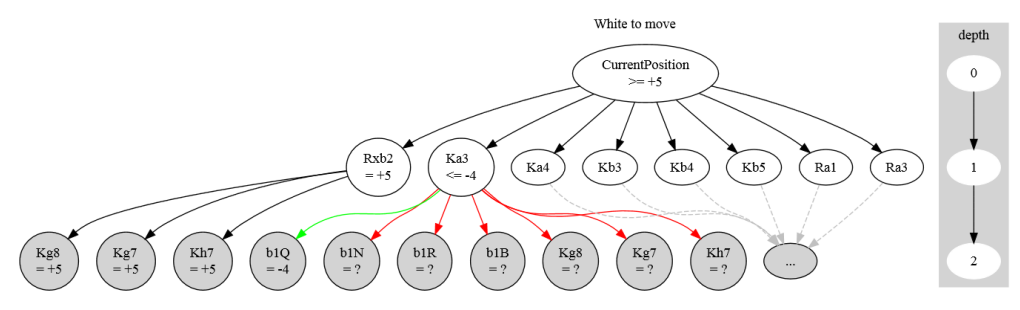
1. Beide Seiten besitzen keine Dame
2. Beide Seiten haben neben der Dame entweder keine weiteren Figuren oder aber maximal eine leichte Figur

2.3 Alpha-Beta-Suche

Bedenkzeit ist im Wettbewerbsschach eine begrenzte Ressource. Deshalb muss sowohl die Güte eines Schachspielers, als auch die eines Schachcomputers, nicht nur nach der Qualität seiner Züge bewertet werden, sondern auch nach der benötigten Bedenkzeit. Durch das exponentielle Wachstum des Spielbaums gerät man beim Minimax Algorithmus, welcher den gesamten Spielbaum analysiert, schnell an rechnerische Grenzen für Computer. Eine solche Grenze kann bereits bei einer Tiefe von 10 auftreten, da bei dieser Tiefe schon mehr als 69 Billionen Züge analysiert werden müssen [Wikipedia contributors, 2020]. Alpha-Beta Suche wird

verwendet, um diesen Prozess zu beschleunigen ohne die Qualität der Analyse zu beeinträchtigen [Knuth und Moore, 1975].

Der Ansatz der Alpha-Beta-Suche beruht auf der Tatsache, dass eine vollständige Analyse jedes einzelnen Teilpfads des Spielbaums in den meisten Fällen nicht nötig ist. Aus diesem Grund ist es beim Schach möglich, mittels einer Zugvorsortierung bereits bei einer Tiefe von vier bis zu 99% des Spielbaums vernachlässigen zu können [Wikipedia, 2019]. Der Name Alpha-Beta-Suche kommt von den in der Alpha-Beta-Suche verwendeten zwei Variablen Alpha und Beta. Für die beiden Variable gilt:

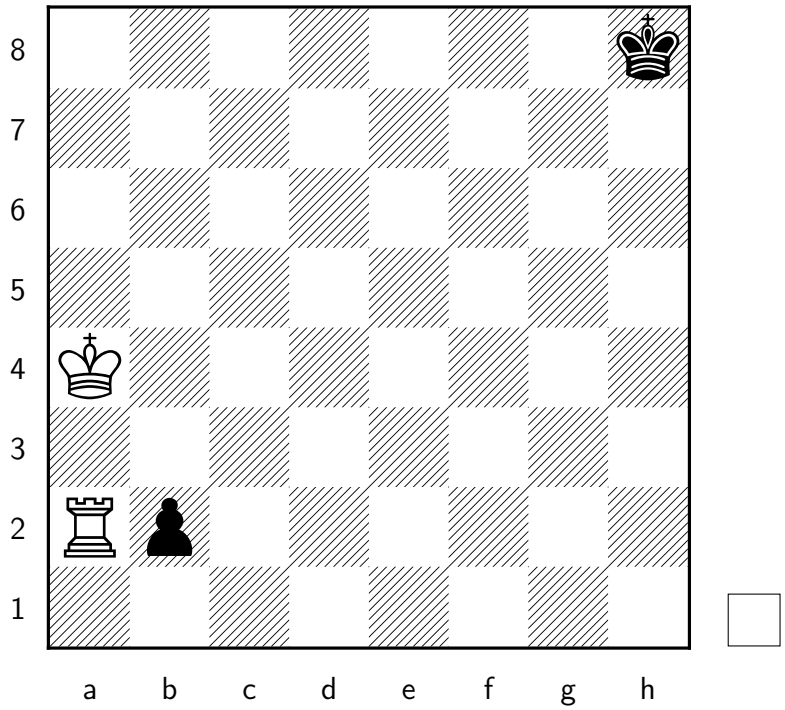


α = Position mit der **momentan** bestmöglich erreichbaren Position für den maximierenden Spieler (Weiß)

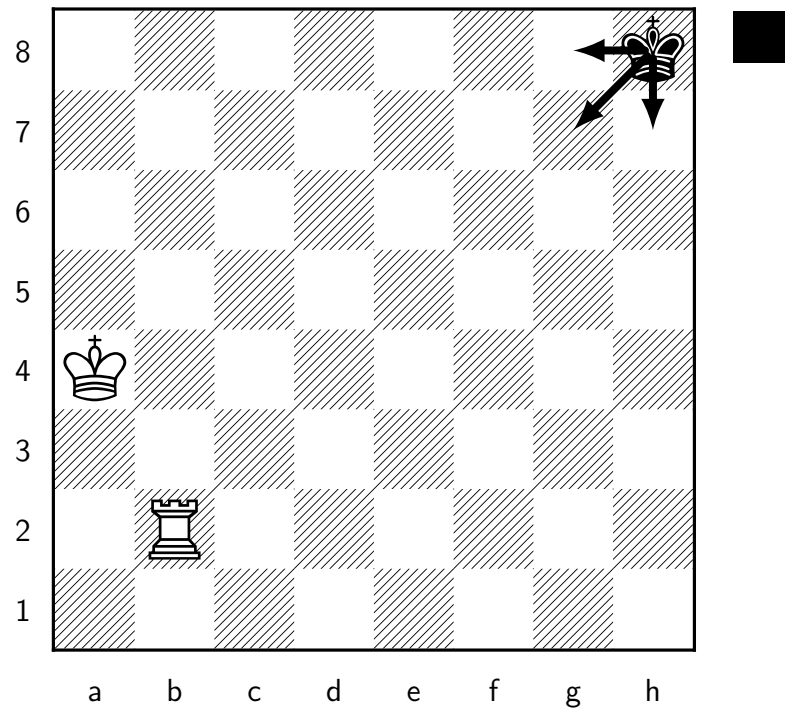
β = Position mit der **momentan** bestmöglich erreichbaren Position für den minimierenden Spieler (Schwarz)

Somit kann beim Analysieren eines Teilbaums potenziell bereits nach der Evaluierung des ersten Blattknotens bemerkt werden, dass diese Zugfolge weniger vorteilhaft ist, als eine bereits vorher betrachtete und somit die restlichen Zugmöglichkeiten des Teilbaums ignoriert werden. Im folgenden Beispiel ist Weiß am Zug und es werden die beiden Zugvarianten Rxb2 und Ka3 des Spielbaums (Abbildung 2) der Tiefe 2 betrachtet.

Die Ausgangssituation sieht wie folgt aus:

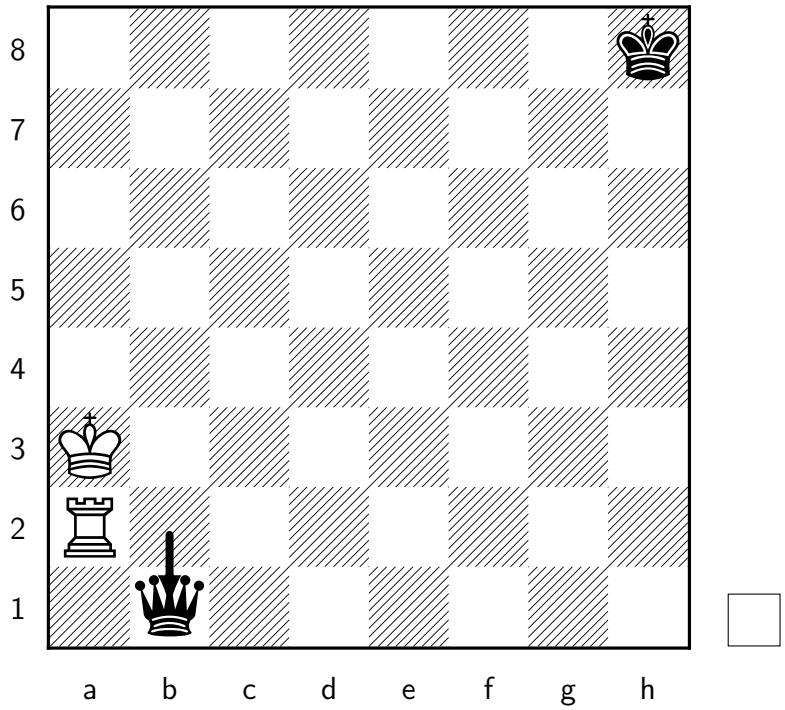


Zunächst wird der Zug Rxb2 analysiert, auf den Schwarz mit drei Zügen reagieren kann (Kg8, Kg7 oder Kh7). Nun sind wir in der maximalen Tiefe angekommen und deshalb werden Kg8, Kg7 und Kh7 der Reihe nach statisch evaluiert. Dies führt jeweils zu einem Wert von +5, denn Weiß hat in allen drei Variationen einen extra Turm.



Jetzt kann man Alpha mit einem Wert von +5 belegen, denn falls Weiß möchte, kann es den Zug Rxb2 spielen und sich eine Evaluation von +5 garantieren.

Anschließend wird der zweite Zug Ka3 analysiert. Schwarz hat sieben Möglichkeiten auf Ka3 zu antworten (b1Q, b1R, b1N, b1B, Kg7, Kh7, Kg8). Da wir wieder die Tiefe zwei erreicht haben, werden die Züge statisch evaluiert. Die Analyse des ersten Zugs b1Q führt zu einer Evaluation von -4, denn Schwarz hat nun eine Dame für weiß' Turm (im Graphen der Abbildung 2 mit grünem Pfeil gekennzeichnet).



Aus dieser Erkenntnis können wir schließen, dass aus Ka8 höchstens eine Position mit der Evaluation von -4 resultiert, da Schwarz der minimierende Spieler ist. Alpha, die Evaluation die der maximierende Spieler Weiß auf jeden Fall erreich kann, beträgt jedoch bereits +5. Deshalb steht fest, dass Weiß Ka3 nicht spielen wird und die restlichen sechs möglichen Züge von Schwarz brauchen auch nicht mehr evaluiert zu werden (im Graphen der Abbildung 2 mit roten Pfeilen gekennzeichnet).

Ähnlich wie in diesem Beispiel, würde auch ein menschlicher Spieler in dem meisten Situationen wohl kaum Zeit dafür verwenden, einen Zug zu analysieren, der dem Gegner eine extra Dame ermöglicht [Paulsen, 2009].

Um die Alpha-Beta-Suche zu implementieren, wird eine weitere Methode benötigt, die nach der Erkenntnis, dass sich das Fortsetzen der Analyse des Teilbaums nicht weiter lohnt, den Algorithmus für diesen Teilbaum stoppt. Hierfür wird der Pseudocode aus dem Kapitel Minimax mit den beiden beschriebenen Variablen Alpha und Beta erweitert.

Diese werden zu Beginn mit den jeweils schlechtmöglichsten Werten für Weiß und Schwarz belegt ($-\infty$ und ∞) und den Prozeduren Minimize (Algorithmus 3) und Maximize (Algorithmus 2) als Argument übergeben.

Algorithmus 2 MAXIMIZE Funktion

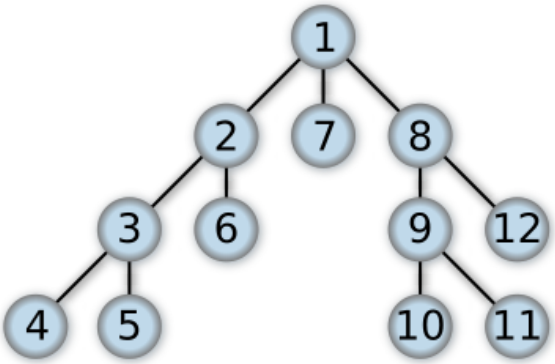
```
1:  function MAXIMIZE(Integer d, Position p, Float alpha, Float beta)
2:      if d < 1 then
3:          return evaluate(p)
4:      else
5:          score ← alpha
6:          legalMoves ← findLegalMoves(p)
7:          for all m in legalMoves do
8:              p.move(m)
9:              moveScore ← minimize(d − 1, p, alpha, beta)
10:             p.undoMove(m)
11:             if moveScore > score then
12:                 score ← moveScore
13:                 if score > alpha then
14:                     alpha ← score
15:                 end if
16:             end if
17:             if score >= beta then
18:                 break
19:             end if
20:         end for
21:     end if
22:     return score
23: end function
```

Algorithmus 3 MINIMIZE Funktion

```
1:  function MINIMIZE(Integer d, Position p, Float alpha, Float beta)
2:      if d < 1 then
3:          return evaluate(p)
4:      else
5:          score ← ∞
6:          legalMoves ← findLegalMoves(p)
7:          for all m in legalMoves do
8:              p.move(m)
9:              moveScore ← maximize(d − 1, p, alpha, beta)
10:             p.undoMove(m)
11:             if moveScore < score then
12:                 score ← moveScore
13:                 if score < beta then
14:                     beta ← score
15:                 end if
16:             end if
17:             if score <= alpha then
18:                 break
19:             end if
20:         end for
21:     end if
22:     return score
23: end function
```

2.3.1 Vorsortieren

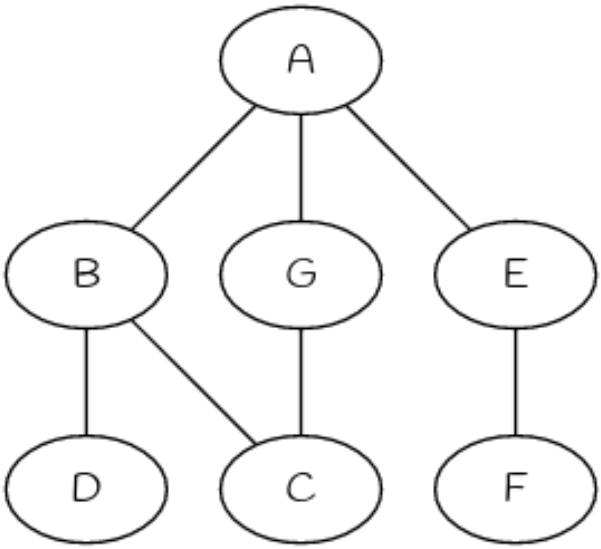
Die Effektivität des Alpha-Beta Algorithmus, der in Kapitel 2.3 beschrieben wurde, ist stark abhängig von der Reihenfolge der Züge, in der diese ausgewertet werden. So ist es beim Alpha-Beta Algorithmus von Vorteil, wenn jene Züge, die für den jeweiligen Spieler vorteilhafter sind, zuerst ausgewertet werden [Russell und Norvig, 2010]. Es existieren verschiedenste Herangehensweisen, dies umzusetzen. Da die möglichen Spielzüge in einem Suchgraphen abgebildet werden können und ein Schachcomputer für den allgemeinen Bedarf möglichst wenig Speicher benötigen sollte, bietet sich das *Depth-First* Suchverfahren (zu dt. etwa *Tiefensuche*) an. Dabei werden zuerst tieferliegende Knoten ausgewertet, bevor benachbarte Knoten ausgewertet werden. Das Depth-First Verfahren bietet den Vorteil, dass nur jeweils ein Pfad des Baumes im Speicher gehalten werden muss [Wiki, 2019a]. Abbildung 3 zeigt eine beispielhafte Evaluierung eines Baumes nach dem Depth-First Verfahren.



Die Zahlen innerhalb der Knoten repräsentieren die Reihenfolge, in der die Knoten besucht werden. Im Beispiel wird zuerst der Wurzelknoten betrachtet. Von diesem ausgehend wir der am weitesten links befindliche Kindknoten evaluiert (markiert durch die Zahl 2). Der nächstfolgende Knoten ist wiederum der am weitesten links befindliche Kindknoten, in der Abbildung markiert durch die Zahl 3. Nachdem der Knoten mit der Zahl 4 evaluiert wurde, ist die maximale Tiefe für diesen Pfad erreicht,

da der Knoten mit der Zahl 4 keine Kindknoten besitzt. Aus diesem Grund wird eine Ebene höher beim Knoten mit der Zahl 3 geschaut, ob dieser noch weitere Kindknoten besitzt. Da dies der Fall ist, werden jene weiteren Kindknoten evaluiert. Im abgebildeten Beispiel betrifft dies ausschließlich den Knoten mit der Zahl 5. Da weder Knoten 5 noch eine Ebene höher Knoten 3 weitere Kindknoten besitzen, wird wieder eine Ebene weiter oben geschaut. Dieser Prozess wird solange fortgeführt, bis alle Knoten besucht wurden.

Weil ein Schachcomputer, der in einem Spiel gegen einen Menschen Einsatz findet, in einer für seinen Gegner vertretbaren Zeit antworten muss, werden Zeitmanagementstrategien benötigt. Diese legen fest, in welcher Reihenfolge die möglichen Spielzüge ausgewerten werden, um die aussichtsreichsten Spielzüge möglichst am Anfang durchzurechnen. Bei den Depth-First Suchverfahren hat sich das *Iterative Deepening* (zu dt. etwa *iterative Tiefensuche*) als grundlegende Zeitmanagementstrategie durchgesetzt. Bei dieser Zeitmanagementstrategie werden alle möglichen Pfade eines Spielbaumes bis zu einer Tiefe von 1 ausgewertet. Ist nach Abschluss der Auswertung die verfügbare Zeit des Spielzugs noch nicht abgelaufen, so wird die Suchtiefe um eins erhöht und eine weitere Evaluierungsphase beginnt [Wiki, 2019b]. Abbildung 4 stellt diesen Vorgang grafisch an einem Beispiel dar.



Bei der folgenden beispielhafter Erläuterung des Iterative Deepening wird angenommen, dass die Knoten von links nach recht abgearbeitet werden. Im ersten Durchlauf werden bei einer Tiefe von 1 nur die Knoten A B G und E evaluiert. Somit sind dem Schachcomputer jetzt bereits alle möglichen Spielzüge bis zu einer Tiefe von eins bekannt, sodass er bei Abbruch des Spielzuges aufgrund der abgelaufenen Zeit bereits eine mögliche gute Entscheidung fällen kann. Darstellung 19 zeigt die Reihenfolge der ausgewerteten Knoten für die Tiefen 2 und 3.

$$\begin{aligned} &\text{Tiefe 2: } A\ B\ D\ C\ G\ C\ E\ F \\ &\text{Tiefe 3: } A\ B\ D\ C\ G\ G\ C\ B\ E\ F \end{aligned} \tag{19}$$

2.3.2 Ruhesuche

Eine weitere Methode den Minimax Algorithmus zu verbessern ist die sogenannte Ruhesuche. Im Schach kommt es oftmals zu erzwungenen Zugsequenzen (engl. *forced line*). Das bedeutet, dass die ziehende Seite eine eingeschränkte Wahl hat zu ziehen. Diese Situation tritt beispielsweise auf, wenn der eigene König im Schach steht oder weil dem Gegner sonst eine vorteilhafte Position ermöglicht wird. Ein Problem entsteht

nun, wenn die Analysetiefe des Spielbaums mitten in einer erzwungenen Zugsequenzen endet, denn dann liefert die Analyse keine aussagekräftige Evaluation.

Es werden deshalb drei Fälle definiert, bei dessen Eintreffen die Standardsuchtiefe erweitert wird [Stuckardt, o. J.]:

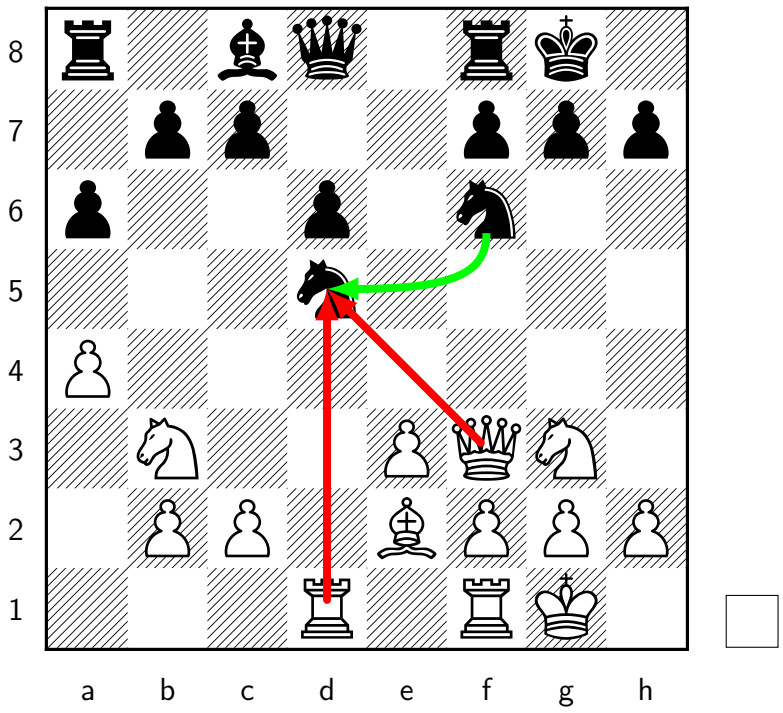
1. Der ziehende Spieler befindet sich im Schach.
2. Der ziehende Spieler kann mit einem Bauern die letzte Reihe erreichen und diesen umwandeln.
3. Der ziehende Spieler kann einen potenziell vorteilhaften Schlagzug tätigen.

Die ersten beiden Kriterien sind selbsterklärend, mit der Ausnahme, dass beim umwandeln der Bauern nur der Fall Dame und der Fall Springer berücksichtigt werden. Grund hierfür ist, dass die Bewegungsmöglichkeiten des Turms und des Läufers beide in denen der Dame enthalten sind. Es gibt zwar Situationen, in denen durch Pattgefahr ein Unterbefördern (engl. *underpromote*) in einen Turm beziehungsweise Läufer sinnvoll ist, allerdings ist das eine Seltenheit und es lohnt sich nicht dafür den Spielbaum aufzublähen [Stuckardt, o. J.]. Das dritte Kriterium wird im Folgenden erläutert.

Erfahrene Schachspieler opfern selten Figuren, ohne dafür etwas im Gegenzug zu erhalten. Aus diesem Grund sind die meisten Schlagzüge im Schach sogenannte Figurenabtausche. Hier schlägt beispielsweise Weiß eine Figur von Schwarz, welche dann wiederum eine Figur von Weiß, meistens mit ähnlichem Wert, schlägt, sodass nach dem Abtausch wieder ein ähnliches Kräftegleichgewicht herrscht wie vor dem Abtausch. Oftmals folgen mehrere Abtausche aufeinander, bei dem dann mehr als nur zwei Figuren geschlagen werden. Ein vorteilhafter Schlagzug ist nun ein Schlagzug, bei dem entweder

1. die geschlagene Figur wertvoller ist als die Figur, welche schlägt, oder

2. die angegriffene Figur von weniger gegnerischen Figuren verteidigt wird, als sie von eigenen Figuren bedroht wird. Dies kann dazu führen, dass der initiale Schlagzug nicht unbedingt einen direkten Vorteil bringt, der gesamte Figurenabtausch jedoch schon [Shannon, 1950][Stuckardt, o. J.].



Kriterium a) ist sehr intuitiv verständlich, da der Tausch einer weniger wertvollen Figur für eine wertvollere per Definition vorteilhaft ist, sofern keine anderen negativen Konsequenzen daraus folgen.

Kriterium b) ist da schon etwas komplizierter. Hier werden auch Schlagzüge betrachtet, welche gleichwertige oder, wie im oberen Beispiel dargestellt, sogar weniger wertvolle Figuren schlagen, solange das Schlagfeld feldbeherrschungstechnisch majorisiert wird [Stuckardt, o. J.].

Im Beispiel sieht dann der Figurenabtausch folgendermaßen aus: Weiß schlägt den gegnerischen Springer auf d5 mit dem Turm auf d1. Anschließend schlägt Schwarz mit dem Springer auf f6 zurück und zuletzt schlägt Weiß diesen Springer mit der Dame auf f3. Am Schluss dieses Abtausches

hat Schwarz zwei Springer (6 Bauern) verloren und Weiß einen Turm (5 Bauern). Der Tausch favorisiert also Weiß, obwohl der initiale Schlagzug zunächst nicht vorteilhaft ist.

Hierbei ist außerdem wichtig zu erwähnen, dass ein Figurentausch nach Kriterium 2. nicht notwendigerweise zu einem favorisierenden Ergebnis für den Spieler mit der Feldbeherrschung führt. Nehmen wir nämlich an, im oben beschriebenen Beispiel steht auf d5 zu Beginn kein schwarzer Springer, sondern ein schwarzer Bauer, so würde der gleiche Abtausch mit einem Verlust von 4 Bauern für Schwarz (Bauer + Springer) und einem Verlust von 5 Bauern für weiß (Turm) resultieren, obwohl sich die Feldbeherrschung nicht geändert hat. Des Weiteren muss die Stellungsdarstellung bereits jene Schlagmöglichkeiten vorher erkennen, die sich im Verlauf des Figurenabtausches offenbaren können. Dies tritt beispielsweise auf, wenn sie vorher durch eigene Figuren blockiert wurden (sogennante *Batterien*) [Stuckardt, o. J.].

Die Ruhesuche wird nun so implementiert, dass bis zu einer maximalen Tiefe von 30 Halbzügen nach einer Ruhestellung, die keine der drei Bedingungen zulässt, gesucht wird. Bei dieser Tiefe wurde empirisch gezeigt, dass die meisten Positionen bereits zu einer Ruhestellung gekommen sind. Die Selektivität von Kriterium a) und b) ist stark genug, sodass gegebenenfalls auftretende Effizienzprobleme vernachlässigbar sind [Stuckardt, o. J.].

2.3.3 Transpositionstabellen

Im Schach treten viele *Transpositionen* auf. Transpositionen sind verschiedene Permutationen einer Zugsequenz, die in die selbe Stellung münden [Russell und Norvig, 2010]. Folgendes Beispiel soll dies näher illustrieren: Weiß hat einen Zug, $a1$, der von Schwarz mit $b1$ beantwortet werden kann, und einen nicht verwandten Zug $a2$ auf der anderen Seite des Bretts, der mit $b2$ beantwortet werden kann. Beide Sequen-

zen $[a1, b1, a2, b2]$ und $[a2, b2, a1, b1]$ münden in der gleichen Stellung. Es ist vorteilhaft, das Ergebnis der ersten Zugsequenz in einer Hashtabelle abzulegen und beim zweiten Mal das Ergebnis aus der Hash Tabelle zu verwenden, anstatt es von Neuem zu berechnen. Das Verwenden von Transpositionstabellen kann einen signifikanten Einfluss haben. So kann die Verwendung dafür sorgen, dass in einigen Fällen der Schachcomputer bis zu der doppelten Tiefe suchen kann [Russell und Norvig, 2010].

In der Hashtabelle wird ein eindeutiger Schlüssel zur Identifikation eines Eintrags verwendet. Im Falle der Transpositionstabellen wird dabei die Stellung als Schlüssel verwendet. Die Stellung muss hierbei mittels einer Hashfunktion in einen Schlüssel umgewandelt werden. Hierfür exisitieren verschiedenste Ansätze und Hashfunktionen. Zu den bekanntesten zählen Zobrist Hashing und BCH Hashing [Wiki, 2020]. Im Zuge dieser Arbeit Zobrist Hashing verwendet. Als eindeutiger Schlüssel wird ein Tupel bestehend aus dem Hash und der Tiefe verwendet.

Als Wert zum Schlüssel wird ein Tripel der Form $< score, alpha, beta >$ gespeichert. Es ist von entscheidender Bedeutung, dass nicht nur der Score abgespeichert wird, sondern auch die dazugehörigen Intervallsgrenzen Alpha und Beta, um überprüfen zu können, ob der Score an der verwendeten Stelle aussagekräftig genug ist [Wiki, 2020]. Liegen das für eine Stellung verwendete Alpha-Beta-Intervall innerhalb eines abgespeicherten Intervalls für diese Stellung oder ist mit dem gespeicherten Intervall identisch, so darf der abgelegte Wert verwendet werden. Andernfalls wird das Intervall angepasst und der Score muss von Neuem berechnet werden.

Da die Menge an Transpositionstabellen zu groß für einen Heimcomputer werden kann, muss eine maximale Anzahl von Transpositionstabellen im Speicher festgelegt werden. Zudem muss ein Verfahren bestimmt und implementiert werden, wann Transpositionstabellen aus dem Speicher entfernt werden. Hierfür eignet sich der Least-Recently-Used (LRU) An-

satz. Das bedeutet, dass, falls die maximale Anzahl an Einträgen erreicht ist, der Eintrag, der am längsten nicht verwendet wird, aus dem Speicher entfernt wird und Platz für den neuen Eintrag schafft. Eine Implementierung dieses Verfahrens in Python ist in Listing 1 dargestellt.

```
class LRUCache:
    def __init__(self, size):
        self.od = OrderedDict()
        self.size = size

    def get(self, key, default=None):
        try:
            self.od.move_to_end(key)
        except KeyError:
            return default
        return self.od[key]

    def __contains__(self, item):
        return item in self.od

    def __len__(self):
        return len(self.od)

    def __getitem__(self, key):
        self.od.move_to_end(key)
        return self.od[key]

    def __setitem__(self, key, value):
        try:
            del self.od[key]
        except KeyError:
            if len(self.od) == self.size:
                self.od.popitem(last=False)
        self.od[key] = value
```

Es handelt sich um einen klassenbasierten Ansatz, wobei intern zur Speicherung ein `OrderedDict` verwendet. Der Vorteile eine `OrderedDict` gegenüber eines normalen Dictionary ist die Tatsache, dass ein `OrderedDict` sich die Reihenfolge, in der neue Schlüssel-Wert-Paare hinzugefügt werden, merkt. An dieser Stelle wird ein Wrapper für das `OrderedDict` verwendet, um den Alterungsprozess von Einträgen kontrollieren zu können. Wird

auf einen bereits vorhandenen Eintrag zugegriffen, so wird dieser erst an das Ende des `OrderedDict` verschoben und dann zurückgegeben. Wird hingegen ein neuer Eintrag hinzugefügt, obwohl die maximale Anzahl an Einträgen bereits erreicht ist, so wird das erste Element im `OrderedDict`, das somit am längsten nicht verwendet wurde, gelöscht und der neue Eintrag am Ende eingefügt.

Die in Listing 1 dargestellte Implementierung eines LRU Caches wird in der späteren Umsetzung der Transpositionstabellen verwendet.

3 Umsetzung

3.1 Tools

Zur Umsetzung der algorithmischen Anteile werden ausschließlich Funktionalitäten der Python Standardbibliothek verwendet. Für die Repräsentation eines Schachspiels sowie die Visualisierung wird das Paket *Python-Chess* verwendet. Das Paket ist im Python Package Index (PyPI) verfügbar. Aus diesem Grund kann es, wie in Listing 2 dargestellt, über den Python Package Installer *pip* installiert werden.

```
$ python -m pip install python-chess
```

Des Weiteren wird zur Dokumentation der Umsetzung und für das Spielen gegen den Schachcomputer ein Jupyter Notebook verwendet.

3.2 Implementierung

Zuerst werden alle notwendigen Pakete und Module importiert. Dabei wird neben den Modulen der Standardbibliothek (`random`, `sys`, `time` und `uuid`) und den Datenstrukturen `defaultdict` und `OrderedDict` aus dem `collections` Modul noch das Paket *Python-Chess* (`chess`) benötigt, welches bereits installiert wurde. Zudem werden für die Anzeige einige Funktionen aus IPython benötigt, die durch die Installation und Verwendung von Jupyter Notebooks bereitgestellt werden.

[ ]:
```
import random
import sys
import time
import uuid

import chess
import chess.engine
import chess.polyglot
import chess.svg

from collections import defaultdict
from collections import OrderedDict

from IPython.display import display, HTML, clear_output
```

Zunächst werden globale Konstanten definiert.

- `QUIESCENCE_SEARCH_DEPTH` ist die maximale Suchtiefe der Ruhesuche.
- `TABLE_SIZE` is die maximale größe der Transpositionstabellen.
- `TIMEOUT_SECONDS` wird verwendet, um die Zeit des Schachcomputers zur Berechnung seines nächsten Zuges festzulegen.

```
[ ]: QUIESCENCE_SEARCH_DEPTH: int = 20
     TABLE_SIZE: int = 1.84e19
     TIMEOUT_SECONDS: int = 30
```

Im folgenden Schritt werden einige globale Variablen definiert. Dieser Schritt ist notwendig, um die Funktionsköpfe möglichst nah am Pseudocode aus dem Theorieteil und der Literatur zu halten.

- Die Variable `best_move` enthält den besten Zug für die aktuelle Iteration der `iterative_deepening`-Funktion.
- Demgegenüber speichert die Variable `global_best_move` den besten Zug aus allen Iterationen der iterativen Tiefensuche und somit den Zug, der vom Schachcomputer am Ende seines Zuges ausgeführt wird.
- `current_depth` speichert die aktuelle Tiefe für das Iterative Deepening.
- `endgame` ist eine Flag, welche auf True gesetzt wird, sobald sich das Spiel im Endspiel befindet.
- `is_timeout` ist genau dann wahr, wenn die Zeit des Schachcomputers abgelaufen ist.
- Bei `move_scores` handelt es sich um ein Dictionary, dass den Score für jeden bereits besuchten Zug im Iterative Deepening Verfahren speichert. Dabei handelt es sich speziell um ein verschachteltes Dictionary. Die Schlüssel auf der obersten Ebene bilden die Stellungen in Forsyth-Edwards Notation (FEN). Der dazugehörige Wert

ist wiederum ein Dictionary mit den Zügen als Schlüssel und dem dazugehörigen Score als Wert.

- `piece_zobrist_values` ist eine Liste von Listen, wobei die inneren Listen jeweils ein Feld repräsentieren und ihrerseits für jede einzelne Figur einen eindeutigen Zobrist Hash beinhalten.
- `repetiton_table` speichert Stellungen um zu überprüfen, ob eine Stellung bereits zwei mal vorgekommen ist.
- In der Variablen `start_time` wird die Startzeit des Zuges des Schachcomputers abgelegt.
- `transposition_table` ist ein LRU Cache mit der maximalen Größe `TABLE_SIZE` und bildet damit die Transpositionstabellen ab.
- `zobrist_turn` ist ein Hash, welcher mit dem Zobrist Hash XOR verrechnet wird, falls die ziehende Person wechselt.

[ ]:
```
best_move = None
current_depth = 0
endgame = False
global_best_move = None
is_timeout = False
move_scores = defaultdict(dict)
piece_zobrist_values = []
repetition_table = {}
start_time = 0.0
transposition_table = None
zobrist_turn = 0
```

Nachdem alle notwendigen globalen Definitionen getätigt worden sind, wird nun die verwendete Bewertungsheuristik umgesetzt. Da sich die Bewertungsfunktion in zwei Teile aufteilt, werden diese getrennt implementiert und am Ende zusammengefügt. Zuerst erfolgt die Umsetzung der Figurenwerte. Hierfür wird ein Dictionary `piece_values` angelegt, dass jeder Figur ihren entsprechenden Wert zuweist.

[ ]:
```
piece_values = {
    chess.BISHOP: 330,
    chess.KING: 20_000,
    chess.KNIGHT: 320,
    chess.PAWN: 100,
    chess.QUEEN: 900,
```

```
    chess.ROOK: 500,
}
```

Die Funktion `get_piece_value` gibt den Figurenwert für eine übergebene Figur auf dem Schachbrett zurück. Handelt es sich bei der Figur um eine der Farbe Schwarz, so wird der negierte Werte zurückgegeben, da der Schwarze Spieler der minimierende Spieler ist.

[ ]:
```
def get_piece_value(piece: chess.Piece) -> int:
    factor = -1 if piece.color == chess.BLACK else 1
    return factor * piece_values.get(piece.piece_type)
```

Als nächstes werden die figurenspezifischen Positionstabellen implementiert. Dafür wird ein Dictionary `piece_squared_tables` angelegt, das für jede Figur die entsprechende Positionstabelle speichert. Die Tabellen werden dabei als Tupel von Tupeln gespeichert, da die Veränderung der Werte während des Spiels nicht zulässig ist. Die Tabellen sind aus dem Theorieteil übernommen worden, weshalb sie aus Sicht des weißen Spielers zu betrachten sind. Das Feld A1 befindet sich somit unten links, was dem Index `[7][0]` entspricht.

[ ]:
```
piece_squared_tables = {
    chess.BISHOP: (
        (-20, -10, -10, -10, -10, -10, -10, -20),
        (-10,   0,   0,   0,   0,   0,   0, -10),
        (-10,   0,   5,  10,  10,   5,   0, -10),
        (-10,   5,   5,  10,  10,   5,   5, -10),
        (-10,   0,  10,  10,  10,  10,   0, -10),
        (-10,  10,  10,  10,  10,  10,  10, -10),
        (-10,   5,   0,   0,   0,   0,   5, -10),
        (-20, -10, -10, -10, -10, -10, -10, -20),
    ),
    chess.KING: (
        (-30, -40, -40, -50, -50, -40, -40, -30),
        (-30, -40, -40, -50, -50, -40, -40, -30),
        (-30, -40, -40, -50, -50, -40, -40, -30),
        (-30, -40, -40, -50, -50, -40, -40, -30),
        (-20, -30, -30, -40, -40, -30, -30, -20),
        (-10, -20, -20, -20, -20, -20, -20, -10),
        ( 20,  20,   0,   0,   0,   0,  20,  20),
        ( 20,  30,  10,   0,   0,  10,  30,  20),
    ),
    chess.KNIGHT: (
        (-50, -40, -30, -30, -30, -30, -40, -50),
```

```
        (-40, -20,   0,   0,   0,   0, -20, -40),
        (-30,   0,  10,  15,  15,  10,   0, -30),
        (-30,   5,  15,  20,  20,  15,   5, -30),
        (-30,   0,  15,  20,  20,  15,   0, -30),
        (-30,   5,  10,  15,  15,  10,   5, -30),
        (-40, -20,   0,   5,   5,   0, -20, -40),
        (-50, -40, -30, -30, -30, -30, -40, -50),
    ),
    chess.PAWN: (
        (  0,   0,   0,   0,   0,   0,   0,   0),
        ( 50,  50,  50,  50,  50,  50,  50,  50),
        ( 10,  10,  20,  30,  30,  20,  10,  10),
        (  5,   5,  10,  25,  25,  10,   5,   5),
        (  0,   0,   0,  20,  20,   0,   0,   0),
        (  5,  -5, -10,   0,   0, -10,  -5,   5),
        (  5,  10,  10, -20, -20,  10,  10,   5),
        (  0,   0,   0,   0,   0,   0,   0,   0),
    ),
    chess.QUEEN: (
        (-20, -10, -10,  -5,  -5, -10, -10, -20),
        (-10,   0,   0,   0,   0,   0,   0, -10),
        (-10,   0,   5,   5,   5,   5,   0, -10),
        ( -5,   0,   5,   5,   5,   5,   0,  -5),
        (  0,   0,   5,   5,   5,   5,   0,  -5),
        (-10,   5,   5,   5,   5,   5,   0, -10),
        (-10,   0,   5,   0,   0,   0,   0, -10),
        (-20, -10, -10,  -5,  -5, -10, -10, -20),
    ),
    chess.ROOK: (
        (  0,   0,   0,   0,   0,   0,   0,   0),
        (  5,  10,  10,  10,  10,  10,  10,   5),
        ( -5,   0,   0,   0,   0,   0,   0,  -5),
        ( -5,   0,   0,   0,   0,   0,   0,  -5),
        ( -5,   0,   0,   0,   0,   0,   0,  -5),
        ( -5,   0,   0,   0,   0,   0,   0,  -5),
        ( -5,   0,   0,   0,   0,   0,   0,  -5),
        (  0,   0,   0,   5,   5,   0,   0,   0),
    ),
}
```

Im Endspiel wird für den König eine andere Tabelle verwendet. Diese ist nachfolgend definiert.

[ ]:
```
kings_end_game_squared_table = (
    (-50, -40, -30, -20, -20, -30, -40, -50),
    (-30, -20, -10,   0,   0, -10, -20, -30),
    (-30, -10,  20,  30,  30,  20, -10, -30),
    (-30, -10,  30,  40,  40,  30, -10, -30),
    (-30, -10,  30,  40,  40,  30, -10, -30),
    (-30, -10,  20,  30,  30,  20, -10, -30),
    (-30, -30,   0,   0,   0,   0, -30, -30),
```

```
    (-50, -30, -30, -30, -30, -30, -30, -50),
)
```

Das Paket Python-Chess weist jeder Positionen einen Ganzzahlwert zu. So erhält das Feld A1 den Zahlenwert 0, das Feld B1 den Wert 1. Diese Zuweisung lässt sich bis zum letzten Feld H8 fortführen, das den Zahlenwert 63 erhält. Um nachfolgend effizient auf die Positionswerte basierend auf dem Zahlenwert des Feldes zugreifen zu können, müssen die Zeilen der Positionstabellen umgekehrt werden, sodass Zeile 1 den Index 0 und die Zeile 8 den Index 7 erhält. Dafür wird im Folgenden über jedes Schlüssel-Wert-Paar des `piece_squared_tables` Dictionary iteriert, die Tabelle in eine Liste konvertiert, diese Liste dann umgekehrt und anschließend in ein Tupel zurück überführt. Analog wird die eine Tabelle, die in der Variablen `kings_end_game_squared_table` abgelegt ist, in eine Liste überführt und deren Elemente in umgekehrter Reihenfolge als Tupel in die Variable zurückgeschrieben.

[ ]:
```
piece_squared_tables = {key: tuple(reversed(list(value)))
                        for key, value in␣
 ↪piece_squared_tables.items()}
kings_end_game_squared_table =␣
 ↪tuple(reversed(list(kings_end_game_squared_table)))
```

Die bisher definierten Positionstabellen sind aus der Sicht des Weißen bzw. maximierenden Spielers definiert. Da der zu entwickelnde Schachcomputer aber beide Farben und damit auch gegen sich selber spielen können soll, müssen noch die Positionstabellen für den minimierenden Spieler generiert werden. Hierzu werden wie beim maximierenden Spieler die Zeilen der Tabellen umgekehrt. Weil sich aber die Tabelle als ganzen um 180° drehen muss, werden zusätzlich alle Spalten, was den Elementen innerhalb der Zeilen entspricht, umgedreht. Das resultierende Dictionary bzw. die resultierende Positionstabelle für den König in der Endphase, werden in den zugehörigen Variablennamen mit dem Präfix `reversed_` abgelegt.

```
[ ]: reversed_piece_squared_tables = {key: tuple([
                                                  piece[::-1]
                                                  for piece in␣
 ↪value][::-1])
                                    for key, value in␣
 ↪piece_squared_tables.items()}
reversed_kings_end_game_squared_table = tuple([
                                               piece[::-1]
                                               for piece in␣
 ↪kings_end_game_squared_table][::-1])
```

Die Funktion `get_piece_quared_tables_value` liefert den Wert der figurenspezifischen Positionstabellen für eine übergebene Figur `piece` auf dem Schachbrett. Der Zeilen- und Spaltenindex in der Tabelle wird basierend auf dem Zahlenwert des Feldes, auf dem die Figur sich befindet (`square`), berechnet. Dabei entspricht der Zeilenindex dem Ergebnis der Ganzzahldivision durch 8 und der Spaltenindex dem Rest aus der Division mit der Zahl 8. Zudem wird ein Parameter `end_game` benötigt, der genau dann wahr ist, wenn sich das Spiel in der Endphase befindet und für den König eine abweichende Positionstabelle verwendet wird. Des Weiteren wird erneut der gefundene Wert negiert, wenn es sich bei der Farbe um Schwarz und somit um den minimierenden Spieler handelt.

```
[ ]: def get_piece_squared_tables_value(piece: chess.Piece,␣
 ↪square: int, end_game: bool = False) -> int:
    factor = -1 if piece.color == chess.BLACK else 1
    row = square // 8
    column = square % 8

    if end_game and piece.piece_type == chess.KING:
        if piece.color == chess.WHITE:
            return␣
 ↪kings_end_game_squared_table[row][column]
        else:
            return␣
 ↪reversed_kings_end_game_squared_table[row][column]

    if piece.color == chess.WHITE:
        piece_squared_table = piece_squared_tables.
 ↪get(piece.piece_type)
    else:
        piece_squared_table =␣
 ↪reversed_piece_squared_tables.get(piece.piece_type)

    return factor * piece_squared_table[row][column]
```

Die Funktion `simple_eval_heuristic` setzt die einfache Bewertungsheuristik um. Sie erhält als Eingabeparameter die aktuelle Stellung `board` und ob sich das Spiel im Endspiel befindet (`end_game`). Die Funktion schaut sich jedes Feld des Schachbretts an. Befindet sich eine Figur auf dem Feld, so werden der Figurenwert und der Positionswert bestimmt und zum Endergebnis, das in der Variablen `piece_value` gespeichert wird, addiert. Das Ergebnis wird dann zurückgegeben.

[ ]:
```
def simple_eval_heuristic(board: chess.Board, end_game:␣
↪bool = False) -> int:
    piece_value = 0
    for square in range(64):
        piece = board.piece_at(square)
        if not piece:
            continue
        piece_value += get_piece_value(piece)
        piece_value +=␣
↪get_piece_squared_tables_value(piece, square, end_game)
    return piece_value
```

Die beiden Dictionaries `zobrist_values_white`und `zobrist_values_black` werden für die Umsetzung des Zobrist Hashing benötigt.

[ ]:
```
zobrist_values_white = {
    chess.PAWN: 1,
    chess.KNIGHT: 2,
    chess.BISHOP: 3,
    chess.ROOK: 4,
    chess.QUEEN: 5,
    chess.KING: 6,
}
zobrist_values_black = {
    chess.PAWN: 7,
    chess.KNIGHT: 8,
    chess.BISHOP: 9,
    chess.ROOK: 10,
    chess.QUEEN: 11,
    chess.KING: 12,
}
```

In der Funktion `intit_zobrist_list()` werden die Zobrist-Schlüssel für jede Figur auf jedem der 64 Schachfelder initialisiert. Es wird `uuid` in Kombination mit dem Operatoren `& (1<<64)-1` verwendet, um einzig-

artige 64 Bit Hashes zu erzeugen. Die erzeugten Hashes werden dann der Reihenfolge aus `zobrist_values_white` und `zobrist_values_black` in die zweidimensionale Liste `piece_zobrist_values` abgespeichert. Anschließend wird die Funktion aufgerufen, um die Liste zu initialisieren.

[ ]:
```
def init_zobrist_list():
    global piece_zobrist_values

    zobrist_turn = uuid.uuid4().int & (1 << 64) - 1
    for i in range(0, 64):
        NO_PIECE = uuid.uuid4().int & (1 << 64) - 1
        WHITE_PAWN = uuid.uuid4().int & (1 << 64) - 1
        WHITE_KNIGHT = uuid.uuid4().int & (1 << 64) - 1
        WHITE_BISHOP = uuid.uuid4().int & (1 << 64) - 1
        WHITE_ROOK = uuid.uuid4().int & (1 << 64) - 1
        WHITE_QUEEN = uuid.uuid4().int & (1 << 64) - 1
        WHITE_KING = uuid.uuid4().int & (1 << 64) - 1
        BLACK_PAWN = uuid.uuid4().int & (1 << 64) - 1
        BLACK_KNIGHT = uuid.uuid4().int & (1 << 64) - 1
        BLACK_BISHOP = uuid.uuid4().int & (1 << 64) - 1
        BLACK_ROOK = uuid.uuid4().int & (1 << 64) - 1
        BLACK_QUEEN = uuid.uuid4().int & (1 << 64) - 1
        BLACK_KING = uuid.uuid4().int & (1 << 64) - 1

        piece_zobrist_values.append(
            [
                NO_PIECE,
                WHITE_PAWN,
                WHITE_KNIGHT,
                WHITE_BISHOP,
                WHITE_ROOK,
                WHITE_QUEEN,
                WHITE_KING,
                BLACK_PAWN,
                BLACK_KNIGHT,
                BLACK_BISHOP,
                BLACK_ROOK,
                BLACK_QUEEN,
                BLACK_KING,
            ]
        )


init_zobrist_list()
```

`zobrist_hash()` wird verwendet, um eine Stellung in einen einzigartigen Zobrist-Schlüssel umzuwandeln. Hierfür wird zunächst die zu überführende Stellung als Parameter (`board`) übergeben. Anschließend wird ein leerer Hash erstellt (`zobrist_hash = 0`), welcher dann den Figu-

ren und Feldern der Stellung entsprechend mit den spezifischen Zobrist-Schlüsseln aus der Liste `piece_zobrist_values` XOR verrechnet wird. Um auf die richtigen Indizes der Figurtypen zuzugreifen, werden die Dictionaries `zobrist_value_white` und `zobrist_value_black` verwendet.

[ ]:
```
def zobrist_hash(board: chess.Board) -> int:
    global zobrist_turn

    zobrist_hash = 0
    if board.turn == chess.WHITE:
        zobrist_hash = zobrist_hash ^ zobrist_turn

    for square in range(64):
        piece = board.piece_at(square)
        if not piece:
            index = 0
        elif piece.color == chess.WHITE:
            index = zobrist_values_white.get(piece.
↪piece_type)
        elif piece.color == chess.BLACK:
            index = zobrist_values_black.get(piece.
↪piece_type)
        zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[square][index]

    return zobrist_hash
```

`zobrist_move()` ändert einen bestehenden Zobrist-Schlüssel (`zobrist_hash`), der die Stellung (`board`) repräsentiert, nachdem ein Zug (`move`) ausgeführt wurde.

- Als erstes wird bestimmt, welcher Spieler am Zug ist, weil diese Information benötigt wird, um zwischen den Dictionaries `zobrist_value_white` und `zobrist_value_black` unterscheiden zu können.
- Danach werden die beiden Felder bestimmt, von wo nach wo die ziehende Figur sich bewegt.
- Anschließend werden die Figurtypen auf den entsprechenden Feldern bestimmt. Falls sich keine Figur auf dem Feld befindet auf das die ziehende Figur hinzieht, so wird `NO_PIECE` verwendet, welcher sich im Array `piece_zobrist_values` immer auf den Positionen `[X][0]` befindet (X in range(0, 64)).

- Um den Zug auszuführen wird die Folgende Reihenfolge von XOR Operationen durchgeführt: Zuerst wird der Figurtyp auf dem Startfeld mit sich selbst verrechnet. Anschließend wird `NO_PIECE` auf diesem Feld verrechnet, weil das Feld nachdem die ziehende Figur wegzieht, von keiner Figur besetzt ist. Danach wird der Figurentyp auf dem Zielfeld mit sich selbst verrechnet. Zuletzt wird der ziehende Figurtyp auf dem Zielfeld verrechnet.
- Am Schluss wird noch die Flag `zobrist_turn` verrechnet, weil nach jedem Zug der ziehende Spieler wechselt.

[ ]:
```
def zobrist_move(board,move,zobrist_hash):
    white_to_move = board.turn
    from_square = move.from_square
    to_square = move.to_square
    moving_piece = board.piece_at(from_square)
    captured_piece = board.piece_at(to_square)

    if white_to_move:
        zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[from_square][zobrist_values_white.
↪get(moving_piece.piece_type)]
        zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[from_square][0]
        if not captured_piece:
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][0]
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][zobrist_values_white.
↪get(moving_piece.piece_type)]
        else:
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][zobrist_values_black.
↪get(captured_piece.piece_type)]
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][zobrist_values_white.
↪get(moving_piece.piece_type)]
    else:
        zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[from_square][zobrist_values_black.
↪get(moving_piece.piece_type)]
        zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[from_square][0]
        if not captured_piece:
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][0]
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][zobrist_values_black.
↪get(moving_piece.piece_type)]
```

```
        else:
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][zobrist_values_white.
↪get(captured_piece.piece_type)]
            zobrist_hash = zobrist_hash ^␣
↪piece_zobrist_values[to_square][zobrist_values_black.
↪get(moving_piece.piece_type)]

    zobrist_hash = zobrist_hash^zobrist_turn
    return zobrist_hash
```

Für die spätere Implementierung der Transpositionstabellen wird ein LRU Cache implementiert repräsentiert durch die Klasse `LRUCache`. Eine detaillierte Beschreibung der Implementierung ist in Kapitel 2.3.3 zu finden.

[ ]:
```
class LRUCache:
    def __init__(self, size):
        self.od = OrderedDict()
        self.size = size

    def get(self, key, default=None):
        try:
            self.od.move_to_end(key)
        except KeyError:
            return default
        return self.od[key]

    def __contains__(self, item):
        return item in self.od

    def __len__(self):
        return len(self.od)

    def __getitem__(self, key):
        self.od.move_to_end(key)
        return self.od[key]

    def __setitem__(self, key, value):
        try:
            del self.od[key]
        except KeyError:
            if len(self.od) == self.size:
                self.od.popitem(last=False)
        self.od[key] = value


transposition_table = LRUCache(TABLE_SIZE)
```

Nachdem die verwendete Bewertungsheuristik und das Zobrist Hashing implementiert sind, kann nun das Alpha-Beta Pruning zusammen mit

dem Iterative Deepening umgesetzt werden. Hierfür wird eine Funktion `iterative_deepening` implementiert, die den Ablauf dieses Prozesses koordiniert. Sie erhält folgende drei Parameter:

- Der Parameter `board` enthält die aktuelle Stellung.
- Die initiale Tiefe, bei der gesucht wird, wird mit dem Parameter `depth` übergeben.
- `color` repräsentiert den Spieler, der aktuell am Zug ist.

Als erstes wird überprüft, ob sich das Spiel aktuell im Endspiel befindet und falls ja, wird die globale Flag `endgame` auf `True` gesetzt. Alle anderen benötigten Informationen werden global abgelegt. Zu Beginn des Iterative Deepening wird geprüft, ob der Spieler nur einen erlaubten Zug machen kann. Ist dies der Fall, so wird er sofort zurückgegeben und keine weiteren Berechnungen sind von Nöten. Ist mehr als ein zulässiger Zug möglich, so wird die aktuelle Systemzeit global gespeichert und die Variable `d` mit 0 initialisiert. `d` repräsentiert hierbei die Tiefe, die zur initialen Tiefe `depth` hinzugefügt wird. In einer Endlosschleife wird sukzessiv die Suchtiefe erhöht und `minimize` bzw. `maximize` aufgerufen. Sowohl `alpha` als auch `beta` werden mit einem sehr hohen bzw. sehr niedrigen Zahlenwert initialisiert. Bei jedem Schleifendurchlauf wird dabei dem globale besten Zug (`global_best_move`) der Zug aus der Variablen `best_move` zugewiesen. Falls der Return Wert `sys.maxsize` oder `-sys.maxsize` entspricht, bedeutet dies, dass ein Matt von einem der beiden Spieler erzwungen werden kann und deshalb wird die Suche hier abgebrochen. Des Weiteren wird überprüft, ob die Zeit des Schachcomputers abgelaufen ist. Ist dies der Fall, wird der globale beste Zug zurückgegeben.

[ ]:
```
def iterative_deepening(board: chess.Board, depth: int,␣
 ↪color: int):
    global best_move
    global current_depth
    global global_best_move
    global is_timeout
```

```
    global start_time
    global endgame

    if not endgame and check_endgame(board):
        endgame = True

    zobrist = zobrist_hash(board)
    increment_repetiton_table(zobrist)

    if board.legal_moves == 1:
        return board.legal_moves[0]

    is_timeout = False
    start_time = time.time()
    d = 0
    current_score = 0

    while True:
        if d > 1:
            global_best_move = best_move
            print(f"Completed search with depth␣
↪{current_depth}. Best move so far: {global_best_move}␣
↪(Score: {current_score})")
        if current_score == sys.maxsize or current_score␣
↪== -sys.maxsize:
            return global_best_move
        current_depth = depth + d
        if color == chess.BLACK:
            current_score = minimize(board,␣
↪current_depth, -sys.maxsize, sys.maxsize, color,␣
↪zobrist)
        else:
            current_score = maximize(board,␣
↪current_depth, -sys.maxsize, sys.maxsize, color,␣
↪zobrist)
        d += 1
        if is_timeout:
            return global_best_move
```

Die Funktion `check_triple_repetition` überprüft, ob die aktuelle Begegnung mit einer Stellung, die dritte ist. In diesem Fall wird `True` zurückgegeben. Ansonsten wird `False` zurückgegben. Als Argument bekommt die Funktion den Zobrist Hash der aktuellen Stellung. Die Funktion ist notwendig, weil die von *python-chess* definierte Funktion `board.can_claim_threefold_repetition()` immer den gesamten Move-Stack aufrollt, um Wiederholungen zu erkennen und somit sehr ineffizient ist.

```
[ ]: def check_triple_repetition(zobrist_hash):
         global repetition_table
         if  zobrist_hash in repetition_table:
             times_encountered = repetition_table[zobrist_hash]
             if times_encountered > 2:
                 return True
             else:
                 return False
         else:
             return False
```

`increment_repetition_table` wird verwendet um den Eintrag mit dem Schlüssel `zobrist_hash` im Dictionary um eins zu erhöhen.

```
[ ]: def increment_repetiton_table(zobrist_hash: int):
         if  zobrist_hash in repetition_table:
             times_encountered = repetition_table[zobrist_hash]
             times_encountered = times_encountered + 1
             repetition_table[zobrist_hash] = times_encountered
         else:
             repetition_table[zobrist_hash] = 1
```

`decrement_repetition_table` wird verwendet um den Eintrag mit dem Schlüssel `zobrist_hash` im Dictionary um eins zu vermindern. Falls der Wert für `times_encountered` eins sein sollte, wird der Eintrag aus dem Dictionary gelöscht.

```
[ ]: def decrement_repetition_table(zobrist_hash: int):
         times_encountered = repetition_table[zobrist_hash]
         if times_encountered == 1:
             del repetition_table[zobrist_hash]
         else:
             times_encountered = times_encountered - 1
             repetition_table[zobrist_hash] = times_encountered
```

Die Funktion `check_endgame` überprüft ob sich eine Stellung im Endspiel befindet. Dafür bekommt die Funktion eine Stellung (`board`) als Parameter. Anschließend wird über die Felder des Bretts iteriert und alle Figuren bis auf Bauern und Könige gezählt. Sobald der Zähler für die Figuren (`counter`) größer als 4 ist, ist sicher, dass sich das Spiel nicht im Endspiel befindet und es wird `False` zurückgegeben. Sollte der Zähler nach durchlaufen der Schleife immer noch kleiner als 4 sein, so befindet sich das Spiel im Endspiel und es wird `True` zurückgegeben.

```
[ ]: def check_endgame(board: chess.Board) -> bool:
         counter = 0
         for square in range(64):
             piece = board.piece_at(square)
             if not piece:
                 continue
             if piece.piece_type is not chess.PAWN and piece.
     ↪piece_type is not chess.KING:
                 counter += 1
                 if counter > 4:
                     return False
         return True
```

Die Implementierung der Funktion `maximize` orientiert sich stark am Pseudocode aus dem Theorieteil. Die Funktion erhält fünf Parameter.

- `board` enthält die aktuelle Stellung.
- `depth` repräsentiert die Tiefe, mit der `maximize` aufgerufen wird.
- `alpha` repräsentiert die Variable Alpha des Alpha-Beta Prunings.
- `beta` repräsentiert die Variable Beta des Alpha-Beta Prunings.
- `zobrist` enthält den Zobrist Hash der aktuellen Stellung

Zu Beginn wird geprüft, ob die Zeit des Schachcomputers bereits abgelaufen ist. Ist dies der Fall, so wird die Variable `is_timeout` auf `True` gesetzt und `alpha` zurückgegeben. Anschließend wird geprüft ob sich die Stellung in einem Schachmatt `board.is_checkmate()` oder einem Patt `board.is_stalemate` befindet, oder ob sich ein Unentschieden über die dreifache Wiederholung einer Stellung erzwingen lässt. Falls einer dieser drei Fälle zutrifft, wird sofort der entsprechende Score (`sys.maxsiz` bzw. `-sys.maxsize` und `0`) zurückgegeben Danach wird überprüft, ob für den `zobrist_hash` der aktuellen Stellung und der aktuellen Tiefe `depth` ein Eintrag in der Transpositionstabelle `transposition_table` vorhanden ist. Ist dies der Fall, so werden anschließend die aktuellen Alpha und Beta Werte mit den Alpha und Beta Werten aus der Transpositionstabelle verglichen. Falls sich das aktuelle Alpha-Beta Intervall innerhalb des Alpha-Beta Intervalls der Transpositionstabelle befindet, wird einfach der `score` der Transpositionstabelle als return Wert wiedergegeben. Sonst wird das Minimum von beiden Alpha-Werten und das Maximum von beiden Beta

Werten bestimmt und diese als aktuelle Alpha und Beta Werte übernommen. Anschließend wird überprüft, ob es sich bei der aktuellen Suchtiefe um eine Tiefe kleiner als 1 handelt. Ist dies der Fall, wird die aktuelle Stellung mittels der Ruhesuche evaluiert und der entsprechende Wert zurückgeliefert. Weil noch kein Zug durchgeführt wurde, wird die Ruhesuche für `maximize` aufgerufen (`quiescnece_search_maximize`). Dabei ist zu beachten, dass die Suchtiefe um eins erhöht wird. Ist die aktuelle Suchtiefe nicht kleiner als eins, so wird als nächstes der Variablen `score` der Wert für `alpha` zugewiesen. Anschließend werden alle zulässigen Züge nach ihren Scores sortiert, sodass aussichtsreichere Züge zuerst evaluiert werden. Ist kein Score für einen zulässigen Zug vorhanden, so werden die Züge mit einem neutralen Score von 0 belegt. Nachfolgend wird über alle zulässigen Züge iteriert und jeweils die Zobrist Hashes für die neu entstehenden Stellungen mit `zobrist_move` berechnet und den folgenden Aufrufen der `minimize` Funktion als Parameter übergeben. Die Variablen `score` und `alpha` werden basierend auf dem Wert von `move_score` aktualisiert, wie dies im Pseudocode aus dem Theorieteil der Fall ist. Zusätzlich wird `best_move` der aktuelle Zug zugewiesen, wenn folgende zwei Bedingungen erfüllt sind:

1. Der aktuelle Score ist größer als Alpha, wodurch die Variable `alpha` aktualisiert wird und ein möglicher neuer bester Zug gefunden wurde.
2. Die aktuelle Suchtiefe entspricht der Suchtiefe, mit der die Funktion `iterative_deepening` die aktuelle Iteration anstieß.

Bevor letztlich der Score der aktuellen Stellung zurückgeliefert wird, wird Stellung und Score mit den aktuellen Alpha und Beta Werten in der Transpositionstabelle abgespeichert. Während des Schleifendurchlaufs wird der `move_score` für einen Zug im Dictionary `move_scores` abgelegt. Dabei wird er der Stellung zugewiesen, die als Ausgangsstellung für den Zug dient.

[ ]:
```
def maximize(board: chess.Board, depth: int, alpha: int,␣
↪beta: int, color: int, zobrist: int) -> int:
    global is_timeout
    global start_time
    global best_move
    global global_best_move

    if time.time() - start_time > TIMEOUT_SECONDS:
        is_timeout = True
        return alpha


    if board.is_checkmate():
        return -sys.maxsize
    if board.is_stalemate():
        return 0
    if check_triple_repetition(zobrist):
        return 0

    if (zobrist, depth) in transposition_table:
        score, a, b = transposition_table[(zobrist,␣
↪depth)]
        if a <= alpha and beta <= b:
            return score
        else:
            alpha = min(alpha, a)
            beta = max(beta, b)

    if depth < 1:
        return quiescence_search_maximize(board, alpha,␣
↪beta, 1)

    score = alpha

    board_scores = move_scores.get(board.fen(), dict())
    moves = sorted(
        board.legal_moves, key=lambda move: -board_scores.
↪get(move, 0),
    )

    for move in moves:
        new_zobrist = zobrist_move(board, move, zobrist)
        increment_repetiton_table(new_zobrist)
        board.push(move)
        move_score = minimize(board, depth - 1, score,␣
↪beta, color, new_zobrist)
        move_scores[board.fen()][move] = move_score
        decrement_repetition_table(new_zobrist)
        board.pop()

        if move_score > score:
            score = move_score
            if depth == current_depth:
                best_move = move
            if score >= beta:
                break
```

```
    transposition_table[(zobrist, depth)] = score, alpha,␣
↪beta
    return score
```

Die Funktion `minimize` ist zu großen Teilen mit der Funktion `maximize` identisch, wie dies auch im Pseudocode aus dem Theorieteil der Fall ist. Die Unterschiede zur `maximize`-Implementierung sind:

- Die Liste der zulässigen Züge ist aufwärts und nicht abwärts sortiert. Somit fängt der Schachcomputer mit dem Zug an, der den geringsten Score besitzt. Zudem ist der für den Spieler schlechteste Zug nun `sys.maxsize`.
- Es wird die Variable `beta` aktualisiert und nicht die Variable `alpha`. Diese wird zudem aktualisiert, wenn der aktuelle `score` niedriger und somit besser ist.

[ ]:
```
def minimize(board: chess.Board, depth: int, alpha: int,␣
↪beta: int, color: int, zobrist) -> int:
    global best_move
    global global_best_move
    global is_timeout
    global start_time

    if time.time() - start_time > TIMEOUT_SECONDS:
        is_timeout = True
        return beta

    if board.is_checkmate():
        return sys.maxsize
    if board.is_stalemate():
        return 0
    if check_triple_repetition(zobrist):
        return 0

    if (zobrist, depth) in transposition_table:
        score, a, b = transposition_table[(zobrist,␣
↪depth)]
        if a <= alpha and beta <= b:
            return score
        else:
            alpha = min(alpha, a)
            beta = max(beta , b)

    if depth < 1:
        return quiescence_search_minimize(board, alpha,␣
↪beta, 1)

    score = beta
```

```
    board_scores = move_scores.get(board.fen(), dict())
    moves = sorted(
        board.legal_moves, key=lambda move: board_scores.
↪get(move, 0),
    )

    for move in moves:
        new_zobrist = zobrist_move(board, move, zobrist)
        increment_repetiton_table(new_zobrist)
        board.push(move)
        move_score = maximize(board, depth - 1, alpha,␣
↪score, color, new_zobrist)
        move_scores[board.fen()][move] = move_score
        decrement_repetition_table(new_zobrist)
        board.pop()

        if move_score < score:
            score = move_score
            if depth==current_depth:
                best_move = move
            if score <= alpha:
                break

    transposition_table[(zobrist, depth)] = score, alpha,␣
↪beta
    return score
```

Die Funktion `quiescence_search_maximize` implementiert die Ruhesuche aus der Sicht des maximierenden Spielers. Sie ist daher weitesgehend identisch mit der Funktion maximize und verfügt auch über die selbe Parameterliste. Ist die maximale Tiefe für die Ruhesuche (`QUIESCENCE_SEARCH_DEPTH`) erreicht, so wird die aktuelle Stellung mittels der Bewertungsheuristik evaluiert und das Ergebnis zurückgeliefert. Bei den Zügen werden nur die, die als favorisierend betrachtet werden (`favorable_moves`), berücksichtigt.

[ ]:
```
def quiescence_search_maximize(board: chess.Board, alpha,␣
↪beta, currentDepth: int):
    global best_move
    global global_best_move
    global endgame

    if currentDepth == QUIESCENCE_SEARCH_DEPTH:
        return simple_eval_heuristic(board, endgame)

    favorable_moves = []
    moves = board.legal_moves
```

```
    for move in moves:
        if is_favorable_move(board, move):
            favorable_moves.append(move)

    if favorable_moves == []:
        return simple_eval_heuristic(board)

    score = alpha
    for move in favorable_moves:
        board.push(move)
        move_score = quiescence_search_minimize(board,␣
↪score, beta, currentDepth + 1)
        board.pop()
        if move_score > score:
            score = move_score
            if score >= beta:
                break

    return score
```

Analog wird die Funktion für die minimierende Ruhesuche `quiescence_search_min` implementiert.

[ ]:
```
def quiescence_search_minimize(board: chess.Board, alpha,␣
↪beta, currentDepth: int):
    global best_move
    global global_best_move
    global endgame

    if currentDepth == QUIESCENCE_SEARCH_DEPTH:
        return simple_eval_heuristic(board, endgame)

    moves = board.legal_moves
    favorable_moves = []

    for move in moves:
        if is_favorable_move(board, move):
            favorable_moves.append(move)

    if favorable_moves == []:
        return simple_eval_heuristic(board)

    score = beta
    for move in favorable_moves:
        board.push(move)
        move_score = quiescence_search_maximize(board,␣
↪alpha, score, currentDepth + 1)
        board.pop()
        if move_score < score:
            score = move_score
            if score <= alpha:
                break

    return score
```

Die Funktion `is_favorable_move` überprüft, ob ein Zug zu einer, wie im Kapitel Ruhesuche beschrieben, vorteilhaften Stellung führt oder nicht. Als Argumente bekommt die Funktion eine Stellung (`board`) und einen Zug (`move`) übergeben. Nun wird überprüft, ob der Zug vorteilhaft ist. Der Zug ist dann vorteilhaft, wenn der Zug ein Schlagzug ist und wenn die schlagende Figur weniger Wert ist als die Figur, welche geschlagen wird oder wenn das Feld von der schlagenden Seite feldbeherrschungstechnisch majorisiert wird. En-Passant Züge werden zuvor schon herausgefiltert, weil in diesem Fall ein Bauer geschlagen wird, welcher sich aktuell auf einem anderen Feld befindet, als das Zielfeld des Schlagzugs.

[ ]:
```
def is_favorable_move(board: chess.Board, move: chess.
↪Move) -> bool:
    if move.promotion is not None:
        return True
    if board.is_capture(move) and not board.
↪is_en_passant(move):
        if piece_values.get(board.piece_type_at(move.
↪from_square)) < piece_values.get(
            board.piece_type_at(move.to_square)
        ) or len(board.attackers(board.turn, move.
↪to_square)) > len(
            board.attackers(not board.turn, move.
↪to_square)
        ):
            return True
    return False
```

Die Funktion `get_opening_data_base_moves` ist eine Hilfsfunktion, die einen Zug aus der Opening Data Base für die aktuelle Stellung (`board`) zurückgibt, falls ein Zug für die übergebene Stellung gefunden wird.

[ ]:
```
def get_opening_data_base_moves(board: chess.Board):
    move = None
    opening_moves = []

    with chess.polyglot.open_reader("Performance.bin") as␣
↪reader:
        for entry in reader.find_all(board):
            opening_moves.append(entry)
```

```
    if opening_moves:
        random_entry = random.choice(opening_moves)
        move = random_entry.move
        print(move)

    return move
```

Für die Anzeige werden die Spielernamen benötigt. Hierfür wird eine Hilfsfunktion `who` implementiert, die einen Spieler als Parameter erhält und den zugehörigen Namen als Zeichenkette zurückgibt.

[ ]:
```
def who(player):
    return "White" if player == chess.WHITE else "Black"
```

Nachdem der Schachcomputer implementiert ist, wird nun die Schnittstelle für den menschlichen Spieler implementiert. Hierfür wird zuerst eine Hilfsfunktion `get_move` benötigt, die die Nutzereingabe in einen Zug umwandelt, sofern dies möglich ist. Zudem wird in ihr überprüft, ob das Spiel vorzeitig zu beenden ist. Dies ist der Fall, wenn der Nutzer als Zug ein `q` (für engl. *quit*) eingibt. Sie erhält als Parameter den dem Nutzer anzuzeigenen Text für die Zugeingabe.

[ ]:
```
def get_move(prompt):
    uci = input(prompt)
    if uci and uci[0] == "q":
        raise KeyboardInterrupt()

    try:
        chess.Move.from_uci(uci)
    except:
        uci = None

    return uci
```

Die Funktion `human_player` repräsentiert den menschlichen Spieler und koordiniert die Züge des Spielers. Dafür wird zuerst die aktuelle Stellung mittels der IPython-Funktion `display` angezeigt. Anschließend werden dem menschlichen Spieler alle zulässigen Züge für die aktuelle Stellung angezeigt und er wird aufgefordert, seinen Zug zu tätigen. Diese Aufforderung geschieht solange, bis der Nutzer einen gültigen Zug eingegeben hat oder aber das Abbruchkriterium `q`.

```
[ ]: def human_player(board):
         display(board)
         uci = get_move(f"{who(board.turn)}'s move [q to␣
     ↪quit]>")
         legal_uci_moves = [move.uci() for move in board.
     ↪legal_moves]

         while uci not in legal_uci_moves:
             print(f"Legal moves: {(', '.
     ↪join(sorted(legal_uci_moves)))}")
             uci = get_move(f"{who(board.turn)}'s move [q to␣
     ↪quit]>")

         return uci
```

Nachdem alle notwendigen Funktionen für den Schachcomputer implementiert sind, kann nun die Funktion `ai_player` implementiert werden. Sie repräsentiert den Schachcomputer dem Spiel gegenüber. Die Funktion besitzt zwei Parameter:

- `board` enthält die aktuelle Stellung.
- `color` repräsentiert den Spieler, der am Zug ist.

Zuerst wird in der verfügbaren Opening Data Base geschaut, ob ein Zug für die übergebene Stellung vorhanden ist. Ist dies der Fall, so wird der Zug zurückgegeben. Andernfalls wird das globale `move_scores` Dictionary zurückgesetzt und die Funktion `iterative_deepening` aufgerufen und deren gewählter Zug zurückgegeben. Es ist zu beachten, dass wenn ein Zug aus der Opening Data Base verwendet wird, die globalen Dictionaries für die Zugscores und die Nachbarstellungen zurückgesetzt werden, da ansonsten keine ausreichende Kontrolle über die Tiefen existiert. Diese Gegebenheit kann durch eine komplexere Implementierung der Zugsortieren optimiert werden, wird an dieser Stellung aber nicht weiter betrachtet.

```
[ ]: def ai_player(board: chess.Board, color: int):
         move = get_opening_data_base_moves(board)

         if move:
             moves_scores = defaultdict(dict)
             return move
```

```
    else:
        return iterative_deepening(board, 0, color)
```

`play_game` ist die Funktion, die den Spielablauf koordiniert. Der Funktion kann optional ein Parameter `pause` übergeben werden. Dieser legt fest, wie lange die Stellung und der zu ihr geführte Zug angezeigt werde soll, bevor der andere Spieler am Zug ist. Der übergebene Zahlenwert wird als Anzahl an Sekunden interpretiert. Nach der Initialisierung des Schachbretts sind solange beide Spieler abwechselnd am Zug, bis das Spiel als beendet angesehen wird oder das Spiel abgebrochen wurde. Anschließend werden je nach Ausgang des Spiels unterschiedliche Ergebnisse angezeigt.

[ ]:
```
def play_game(pause=1):
    board = chess.Board()

    try:
        while not board.is_game_over(claim_draw=True):
            if board.turn == chess.WHITE:
                move = board.
↪parse_uci(human_player(board))
            else:
                move = ai_player(board, chess.BLACK)

            name = who(board.turn)
            board.push(move)
            html = "<br/>%s </b>"%(board._repr_svg_())
            clear_output(wait=True)
            display(HTML(html))
            time.sleep(pause)
    except KeyboardInterrupt:
        msg = "Game interrupted"
        return None, msg, board

    result = None
    if board.is_checkmate():
        msg = "checkmate: " + who(not board.turn) + "␣
↪wins!"
        result = not board.turn
    elif board.is_stalemate():
        msg = "draw: stalemate"
    elif board.is_fivefold_repetition():
        msg = "draw: fivefold repetition"
    elif board.is_insufficient_material():
        msg = "draw: insufficient material"
    elif board.can_claim_draw():
        msg = "draw: claim"
```

```
    print(msg)
    return result, msg, board
```

[ ]:
```
play_game()
```

4 Auswertung und Diskussion

In diesem Kapitel erfolgt eine Einschätzung der Stärke des entwickelten Schachcomputers. Zudem werden vernachlässigte Aspekte benannt und teilweise verwendete Vereinfachungen dargelegt.

Nach Einschätzung der Autoren besitzt der implementierte Schachcomputer eine Elo-Wert von 1400 bis 1500. Diese Einschätzung stammt von Spielen der Autoren gegen den Schachcomputer. Da im vorgegebenen Zeitrahmen keine Spiele gegen mehrere Spieler oder Turniere gegen andere Schachcomputern mit bekannten Elo-Zahlen durchgeführt werden konnten, ist eine genauere oder objektivere Schätzung nicht möglich.

Die Implementierung wurde ausschließlich in der Programmiersprache Python vorgenommen. Die Berechnung in größere Tiefen und der damit einhergehende Anstieg der Stärke des Schachcomputers hätten durch eine Auslagerung in Sprachen wie C ermöglicht werden können. Zudem würde sich bereits die Verwendung von mehreren Threads oder Prozessen positiv auf die Berechnungstiefe auswirken. Auch wirkt sich die Verwendung eines Jupyter Notebooks mit dem mitgelieferten Kernel negativ auf die Ausführung aus. Diese Punkte wurden aufgrund der Aufgabenstellung und der Lesbarkeit des Programmcodes vernachlässigt. Bei der Zugvorsortierung handelt es sich um eine naive Implementierung. Eine auf den Schachcomputer zugeschnittene Implementierung ist in künftigen Untersuchungen vorzuziehen, die auch das unendliche Anwachsen des dazugehörigen Dictionaries verhindert.

Nach langem Testen wurde sich dafür entschieden die maximale Tiefe der Ruhesuche auf 20 zu setzen und Züge, welche den Gegner in Schach setzen nicht als vorteilhaften Zug zu bewerten. Grund für beide Maßnahmen sind die hohen Rechenkosten die entstehen, wenn der Baum bis Tiefe 30 abgesucht wird und der Baum durch viele Möglichkeiten des Schachsetzens aufgebläht wird. Da die gesamte Umsetzung im Ju-

pyter Notebook schon nicht sehr performant ist, mussten deshalb diese Entscheidungen getroffen werden, um nicht zu viel an Performanz einzubüßen.

Literatur

Knuth, D. & Moore, R. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*, 293–326.

Paulsen, P. (2009, 8. Juli). *Lernen unterschiedlich starker Bewertungsfunktionen aus Schach-Spielprotokollen*. TU Darmstadt. Verfügbar 7. November 2019 unter `http://www.ke.tu-darmstadt.de/lehre/arbeiten/diplom/2009/Paulsen_Philip.pdf`

Russell, S. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach, 3rd Edition*. Pearson.

Schach-Club Kreuzberg. (o. J.). *Schach: Immer noch beliebt*. Verfügbar 9. April 2020 unter `http://sparkassen-jugend-open.de/schach-immer-noch-beliebt/`

Schachverein Gifhorn. (o. J.). *Wie das Schachspiel erfunden wurde*. Verfügbar 9. April 2020 unter `https://schachverein-gifhorn.jimdofree.com/vereinsinfos/schach-geschichte/`

Shah, R. B. (2007). *minimax, In Dictionary of Algorithms and Data Structures*. Verfügbar 20. Dezember 2019 unter `https://xlinux.nist.gov/dads/HTML/minimax.html`

Shannon, C. (1950). XXII. Programming a computer for playing chess. *Philosophical Magazine*, *41*, 256–275.

Stuckardt, R. (o. J.). *Ruhesuche (Quiescence Search)*. Verfügbar 21. Januar 2020 unter `http://www.stuckardt.de/index.php/schachengine-fischerle/suchtechniken-im-detail/72-ruhesuche-quiescence-search.html`

Wiki, C. P. (2018). *Simplified Evaluation Function*. Verfügbar 29. Oktober 2019 unter `https://www.chessprogramming.org/Simplified_Evaluation_Function`

Wiki, C. P. (2019a). *Depth-First*. Verfügbar 21. Januar 2020 unter `https://www.chessprogramming.org/Depth-First`

Wiki, C. P. (2019b). *Iterative Deepening*. Verfügbar 21. Januar 2020 unter `https://www.chessprogramming.org/Iterative_Deepening`

Wiki, C. P. (2020). *Transposition Table*. Verfügbar 1. März 2020 unter `https://www.chessprogramming.org/Transposition_Table`

Wikipedia. (2018). 50-Züge-Regel — Wikipedia, Die freie Enzyklopädie. Verfügbar 1. Februar 2020 unter `https://de.wikipedia.org/w/index.php?title=50-Z%C3%BCge-Regel&oldid=183817296`

Wikipedia. (2019). Alpha-Beta-Suche — Wikipedia, Die freie Enzyklopädie. Verfügbar 1. Februar 2020 unter `https://de.wikipedia.org/w/index.php?title=Alpha-Beta-Suche&oldid=191701260`

Wikipedia. (2020). Minimax-Algorithmus — Wikipedia, Die freie Enzyklopädie. Verfügbar 1. Februar 2020 unter `https://de.wikipedia.org/w/index.php?title=Minimax-Algorithmus&oldid=195739275`

Wikipedia contributors. (2020). Shannon number — Wikipedia, The Free Encyclopedia. Verfügbar 1. Februar 2020 unter `https://en.wikipedia.org/w/index.php?title=Shannon_number&oldid=939493351`

Glossar

Perfekte Information Ist ein Begriff der mathematischen Spieltheorie. Demnach besitzt ein Spiel perfekte Information, wenn jedem Spieler zum Zeitpunkt einer Entscheidung stets das vorangegangene Spielgeschehen bekannt ist.