

# Entwicklung einer künstlichen Intelligenz für das Spiel “Schach” in Python

Studienarbeit

des Studiengangs Angewandte Informatik  
an der Dualen Hochschule Baden-Württemberg Mannheim

von

**Lars Dittert & Pascal Schröder**

30.04.2019

**Bearbeitungszeitraum**  
**Matrikelnummer, Kurs**  
**Ausbildungsfirma**  
**Betreuer**

17.09.2018 - 30.04.2019  
5388171 & 5501463, TINF16AI-BC  
IBM Deutschland GmbH, Mannheim  
Prof. Dr. Karl Stroetmann

**Unterschrift Betreuer**

---

# Inhaltsverzeichnis

<b>Erklärung der akademischen Aufrichtigkeit</b>	<b>I</b>
<b>Abstract</b>	<b>II</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Hinführung zum Thema . . . . .	1
1.2 Zweck und Ziel dieser Arbeit . . . . .	2
<b>2 Theoretische Hintergründe</b>	<b>3</b>
2.1 Einordnung der Spieltypen für Schach . . . . .	3
2.2 Geschichte der Spieltheorie . . . . .	6
2.3 Minimax-Algorithmus . . . . .	8
2.4 Alpha-beta pruning . . . . .	14
2.5 Problematik komplexerer Spiele . . . . .	17
2.6 Evaluierungsfunktionen . . . . .	20
<b>3 Technische Grundlagen</b>	<b>32</b>
3.1 Kriterienformulierung . . . . .	32
3.2 Python-Chess Evaluierung . . . . .	32
3.3 Architektur . . . . .	49
<b>4 Implementation</b>	<b>53</b>
<b>5 Evaluation</b>	<b>54</b>
5.1 Kriterienerfüllung . . . . .	54
5.2 Einordnung Intelligenz . . . . .	54
<b>Abkürzungsverzeichnis</b>	<b>III</b>
<b>Appendices</b>	<b>IV</b>

# Erklärung der akademischen Aufrichtigkeit

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema:

*Entwicklung einer künstlichen Intelligenz für das Spiel "Schach" in Python*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.\*

\* falls beide Fassungen gefordert sind

Mannheim, 30.04.2019

Ort, Datum

---

Unterschrift Lars Dittert & Pascal Schröder



# Abstract

# 1 Einleitung

Diese Einleitung dient dazu, in den folgenden Unterkapiteln den Einstieg in das Thema Künstliche Intelligenz und deren Nutzen zu geben. Dabei wird auf die Definition und Eigenschaften eingegangen, sowie die Historie und das Thema der Spieltheorie aufgegriffen. Passend zu der vorgestellten Problematik wird der Bogen zur prüfenden Forschungsfrage gespannt.

## 1.1 Hinführung zum Thema

Künstliche Intelligenz und Machine Learning sind zwei Fachbegriffe der Informatik, die im täglichen Leben immer häufiger auftreten. Besonders in den letzten Jahren tritt Künstliche Intelligenz (KI), oder Artificial Intelligence wie es im englischen heißt, in alle Lebenslagen häufiger auf und selbst ein Informatik-Laie kommt nicht herum mit diesem Begriff konfrontiert zu werden.

Allgemein kann gesagt werden, dass mit Artificial Intelligence versucht wird die Intelligenz eines Menschen nachzubilden und diese auf einen Computer projizieren zu können. Künstliche Intelligenz lässt sich durch vier verschiedenen Merkmale definieren. Aufgelistet haben dies Russell und Norvig in ihrem Buch Artificial Intelligence - A Modern Approach unter den Kategorien menschliches und rationales Denken, sowie menschliches und rationales Handeln.[?]

Auch wenn es so erscheinen mag, weil das Thema KI sehr stark in den Medien präsent ist, ist es jedoch bei weitem keine technische Errungenschaft der letzten Jahren, sondern kann bereits eine Historie über viele Jahre aufweisen. Bereits seit der Entwicklung der ersten Computer besteht der Wunsch darin einem Computer eine eigene Intelligenz zu ermöglichen.

Dabei durchlief KI durch technische Erneuerungen bereits häufiger Hochphasen, während denen das Thema in der Öffentlichkeit eine größere Aufmerksamkeit genoss. Ein Beispiel dafür ist das Programm ELIZA das gegen Ende der 1960er Jahre von Joseph Weizenbaum entwickelt wurde und in dem der Dialog zwischen einem Patienten und dessen Psychotherapeuten simuliert wird.[?]

Die neuste Hochphase wurde vor allem durch das Maschinelle Lernen und neuronale Netze ausgelöst. Jedoch ist ein nicht zu verachtender Aspekt, der der Kunstlichen Intelligenz

zu einer neuern Hochphase verhilft, die Leistungsfähigkeit und der geringere Preis der technischen Geräte.

Hierbei unterscheidet sich die Künstliche Intelligenz in viele unterschiedliche Teilgebiete, wie beispielsweise Robotik oder Machine Learning. Ein weiteres Anwendungsgebiet der Künstlichen Intelligenz ist die sogenannte Game Theory, bei der ein Computer die Regeln eines Spiels erlernt um daraufhin als Gegner dienen zu können. Die Spieltheorie ist ein wesentlicher Bestandteil dieser wissenschaftlichen Arbeit und wird somit im kommenden Kapitel über den Zweck und das Ziel aufgegriffen.

Bereits in den 1950er Jahren befassten sich einige Informatiker mit der Spieltheorie und im speziellen mit dem Spiel Schach. Arthur L. Samuel veröffentlichte 1959 einen wissenschaftlichen Fachaufsatz in dem er sich mit Machine Learning, sowie der Spieltheorie zum Thema Schach befasst hat.[?] Schach bringt dabei den Vorteil mit sich, dass es ein Brettspiel ist, das über einige Regeln verfügt und durch die verschiedenen Figuren und deren möglichen Anordnungen eine gewisse Komplexität mit sich bringt. Durch diese für den Menschen große Komplexität und Schwierigkeit viele Züge in die Tiefe vorhersehen zu können, entstand das Verlangen dies durch eine Künstliche Intelligenz zu realisieren.

## **1.2 Zweck und Ziel dieser Arbeit**

Diese wissenschaftliche Arbeit handelt von der Umsetzung eines Programms, das auf der Basis von KI es ermöglicht als Gegner im Spiel Schach zu dienen. Neben der Implementierung der Schach-KI wird diese Arbeit das nötige Grundwissen vermitteln, das benötigt wird um die in der Umsetzung verwendeten Techniken und Algorithmen zu verstehen.

Hierbei wird ein besonderer Fokus auf den Minimax Algorithmus und die Alpha-Beta-Suche gelegt. Ebenfalls wird eine Einführung in die Geschichte der Spieltheorie gegeben und die Problematik komplexer Spiel aufgezeigt.

## 2 Theoretische Hintergründe

In diesem Kapitel werden die theoretischen Hintergründe zur Spieltheorie im Generellen und vor Allem der Entwicklung von Spiele-Spielenden-Computern eingegangen. Dies wird im Verlauf der Kapitel stets auf das Spiel Schach spezialisiert, da sich der Schwerpunkt dieser Arbeit auf dieses Spiel fokussiert. Zunächst werden die klassischen Spieltypen nach der Spieltheorie betrachtet und Schach in diese Kategorien eingeordnet. Darauf folgend wird als Einführung zu Spiele-KIs (Künstliche Intelligenz) die Geschichte der Entwicklung dieser betrachtet, um Zusammenhänge mit technischen und mathematischen Fortschritten besser zum Verständnis zu bringen.

Anschließend wird speziell auf Algorithmen zur Entwicklung einer Spiele-KI von sequenziellen Nullsummenspielen wie Schach eingegangen. Dabei wird zuerst der grundlegende “Minimax-Algorithmus” betrachtet und zudem auf die Optimierung dieses durch “Alpa-Beta Pruning” eingegangen. Darauf folgend werden zudem die Problematiken der Anwendung dieser auf komplexe Spiele beschrieben und mögliche Lösungen vorgeschlagen.

Abschließend für dieses Kapitel werden durch genannte Problematiken nötig werdende Evaluierungsfunktionen der einzelnen Spielzustände im Spiel Schach genannt und beschrieben, womit die theoretische Grundlagenschaffung für das Entwickeln einer Schach-KI abgeschlossen ist.

### 2.1 Einordnung der Spieltypen für Schach

In der klassischen Spieltheorie werden Spiele anhand von fünf Kategorien unterschieden und eingeordnet. Diese fünf Kategorien sind

- Symmetrische / Asymmetrische Spiele
- Perfekte / Imperfekte Informationslage
- Kooperative / Nicht-kooperativ Spiele
- Simultane / Sequentielle Spiele
- Nullsummenspiele / Nicht-Nullsummenspiel [?]

Diese werden nun alle einzeln beschrieben sowie der Unterschied erläutert. Außerdem wird das Spiel Schach jeweils einer der Spieltypen zugeordnet.

Im ersten Schritt wird zwischen symmetrischen und asymmetrischen Spielen unterschieden. Bei symmetrischen Spielen hat jeder Spieler dieselben Ziele, um das Spiel zu gewinnen. Der Sieg ist dabei abhängig von den gewählten Strategien. Bei asymmetrischen Spielen dagegen verfolgen die teilnehmenden Parteien unterschiedliche Ziele. [?] Ein klassisches Beispiel ist dabei das Spiel des Jägers und des Gejagten - während der Jäger versucht zu den Gejagten zu fangen, versucht der Gejagte zu entkommen. Unter Umständen wird auch der Erfolg im Nachhinein anhand verschiedener Kriterien evaluiert, so dass unter Umständen beide Seiten für sich einen Sieg erringen können.

Schach stellt dabei ein klassisches Beispiel eines symmetrischen Spiels dar. Beide Seiten versuchen jeweils den gegnerischen König Schachmatt zu setzen, ohne dabei selbst zuvor in Schachmatt gesetzt zu werden.

Perfekte und Imperfekte Informationslagen in Spielen kategorisieren Spiele nach dem Wissen, den der Spieler über das Spiel hat. Bei perfekter Informationslage kann der Spieler über alle Züge, sowohl seiner eigenen als auch die des Gegners, Bescheid und kann die Möglichkeiten des Gegners evaluieren, um so selbst eine bessere Entscheidung zu treffen. Bei imperfekter Informationslage ist dies nicht der Fall. [?] Bei vielen Kartenspielen beispielsweise, wie Poker oder Black Jack, hat ein Spieler keine Informationen über die Karten in der Hand der anderen Spieler. Dies ist dann als imperfekte Informationslage zu bezeichnen.

Im Spiel Schach jedoch sind die Figuren und dessen Positionen beider Spieler für alle Seiten offengelegt und einsehbar. Dadurch können sich beide Spieler ein Gesamtbild über die Situation machen, ohne, dass ein Mangel an Information herrscht.

Bei kooperativen Spielen handelt es sich um Spiele, in denen mehrere Spieler zusammen an einem Ziel arbeiten können oder gar müssen, um ihr Ergebnis zu verbessern. Dies kann in Bündnissen münden, die mehrere Spieler zusammen eingehen, um an einem Ziel zu arbeiten, dass alle gemeinsam erreichen können. [?] Ein Beispiel dafür sind Wahlen, die nach Mehrheitsentscheid gewonnen werden können. Dabei gilt es für die Teilnehmer Allianzen zu formen, um gemeinsam eine Möglichkeit durchzusetzen. [?] In Nicht-kooperativen Spielen dagegen ist es verboten oder nicht zielführend Allianzen zu formen, da kein gemeinsamer Sieg errungen werden kann. [?]

Schach stellt dabei ein Nicht-kooperatives Spiel dar, da die Spieler sich nicht miteinander, sondern nur gegeneinander antreten können.



Simultane Spiele sind Spiele, bei denen alle Spieler gleichzeitig Aktionen unternehmen können. Ein Beispiel stellt dabei ein Kampf dar, in der beide Seiten sich gleichzeitig bewegen können und in Echtzeit auf die Bewegungen des Gegners reagieren müssen. Bei sequenziellen Spielen dagegen wechseln sich alle Seiten mit ihren Zügen ab. Dies bedeutet auch, dass die Spieler über die vorangegangenen Züge ihrer Gegner Bescheid wissen. [?]

Wie die meisten Brettspiele stellt Schach dabei ein sequentielles Spiel dar. Die Spieler wechseln sich mit ihren Zügen ab und Wissen über die durch den Gegner bewegten Figuren Bescheid.

Als letzte Unterscheidung gelten Nullsummenspiele und Nicht-Nullsummenspiele. Bei Nullsummenspielen bedeutet ist Gewinn für den einen Spieler immer einhergehend mit einem gleichzeitigen Verlust für einen anderen Spieler. Dies bedeutet, dass Werte für Gewinn und Verluste für alle Spieler zusammen gerechnet immer auf den Wert Null hinauslaufen. Bei Nicht-Nullsummenspielen ist dies nicht der Fall. Dabei können von einer Aktion eines Akteurs mehrere Akteure gleichzeitig profitieren, ohne, dass andere darunter leiden.[?] Ein Beispiel dafür ist das Gefangenendilemma. Dabei können zwei Gefangene jeweils die Entscheidung treffen, ob sie ihr Verbrechen gestehen oder Schweigen. Von ihrem Handeln sowie dem Handeln der anderen Partei hängt die Länge ihrer Strafe ab. [?] Diese ist folgender Tabelle zu entnehmen:

	B schweigt	B gesteht
A schweigt	A: 2 Jahre; B: 2 Jahre	A: 6 Jahre; B: 1 Jahr
A gesteht	A: 1 Jahr; B: 6 Jahre	A: 4 Jahre; B: 4 Jahre

Dies ist als klassisches Nicht-Nullsummenspiel bekannt, da beide Seiten je nach Reaktion zusammengerechnet minimal 4 Jahre und maximal 8 Jahre Strafe bekommen. Somit ist das Gesamtergebnis nicht stets das gleiche und beide Seiten können von gegenseitigem Schweigen profitieren.

Anders ist dies bei Schach. Schach ist als Nullsummenspiel zu betrachten, da das Schlagen einer Figur zwar Gewinn auf der einen Seite auslöst, gleichzeitig aber einen ebenso hohen Verlust auf der anderen Seite, wodurch sich das Gesamtspiel wieder ausgleicht.

Nachdem nun Schach in die Spieltheorie-Typen eingeordnet ist, wird in folgenden Kapiteln näher auf die Möglichkeiten zur Durchführung des Schach Spiels durch einen Computer eingegangen.

## 2.2 Geschichte der Spieltheorie

Die Geschichte der Spieltheorie im Computerumfeld beginnt mit Claude Shannon im Jahre 1949, als dieser seine Gedanken zur möglichen Realisierung eines Schach spielenden Computers veröffentlicht. Dabei begründet er zunächst seine Auswahl auf das Spiel Schach für sein langfristiges Ziel eines spielenden Computers und legt dann das Ziel an sich fest.

“The chess machine is an ideal one to start with, since: (1) the problem is sharply defined both in allowed operations (the moves) and in the ultimate goal (checkmate); (2) it is neither so simple as to be trivial nor too difficult for satisfactory solution; (3) chess is generally considered to require ‘thinking’ for skillful play; a solution of this problem will force us either to admit the possibility of a mechanized thinking or to further restrict our concept of ‘thinking’; (4) the discrete structure of chess fits well into the digital nature of modern computers. . . . It is clear then that the problem is not that of designing a machine to play perfect chess (which is quite impractical) nor one which merely plays legal chess (which is trivial). We would like to play a skillful game, perhaps comparable to that of a good human player.” [?]

Die wesentlichen Punkte von Claude Shannon sind dabei, dass Schach weder zu simpel noch zu komplex ist, um es von einem Computer spielen zu lassen - Es gibt ein eindeutiges Ziel und eindeutige Operationen auf dem Weg zu diesem Ziel, aber kein eindeutig perfektes Spiel. Dementsprechend kann weder dies das Ziel sein, noch kann es das Ziel sein, einfach nur ein legales Spiel durchzuführen. Vielmehr muss es mit dem Schachspiel von Menschen vergleichbar sein. Menschen kennen den perfekten Zug nicht, können jedoch eine begrenzte Zahl an Zügen in dessen Wahrnehmungsbereich evaluieren und den in diesem Rahmen scheinbar besten Zug auswählen. Ein ähnliches Verhalten sollte auch bei einem Computer erkennbar sein.

Der vorgeschlagene Ansatz zum Evaluieren der Züge stellt dabei ein Algorithmus dar, der heute unter “Minimax-Algorithmus” bekannt ist. Dieser Ansatz wird in Kapitel [2.3](#) erklärt.

Anwendung fand dies erstmals im Jahre 1956, als ein Team von Mathematikern und Forschern um Arthur Samuel ein Programm entwickelten, das in der Lage war “Dame” zu spielen. Dies war das erste Mal, das ein Computer in der Lage war in einem strategischen Spiel gegen einen Menschen anzutreten. Im Rahmen dieses Projekts ist auch der Begriff “Artificial Intelligence” oder zu deutsch “Künstliche Intelligenz” entstanden. [?]

Dabei wird vom aktuellen Zustand aus einige Züge vorausgeschaut und bewertet, welcher Zug unter der Annahme, dass auch der Gegner den jeweils besten Zug machen wird, der

aussichtsreichste ist. Zur Berechnung wird ein Minimax Baum ausgehend von der aktuellen Spielsituation aus erstellt.

Auch wurden hier erste Machine-learning Algorithmen verwendet, die dem Computer innerhalb kürzester Zeit das Spiel so gut beibringen, dass dieser dazu in der Lage ist menschliche Spieler zu schlagen. Samuel verwendete dazu zwei Methoden - Einerseits “rote-learning”, das die Werte von bestimmten Spielzuständen, die bereits evaluiert wurden, abspeichert, so dass die Berechnung kein weiteres Mal durchgeführt werden muss. Zum anderen “learning-by-generalization”, das die Parameter der Evaluierungsfunktionen basierend auf vorherigen Spielen anpasst. Dies geschah mit dem Ziel, den Unterschied zwischen dem berechneten Evaluierungswert des Zustands und dem tatsächlichen, auf den Ausgang des jeweiligen Spiels basierenden Werts zu minimieren. [?]

Nur ein Jahr später fand die Theorie von Claude Shannon auch erstmals direkt im Spiel Schach Anwendung. Ein Team um Mathematiker Alex Bernstein entwickelte eine voll funktionale Schach- KI, ebenfalls basierend auf den Minimax Algorithmen. [?] Die Problematik beim Schach, im Vergleich zu anderen Spielen, ist die enorme Anzahl an verschiedenen Zügen und Spielzuständen, die eine Evaluierung jedes einzelnen, selbst bei heutiger Rechenkapazität, unmöglich macht. Dies ist genauer in Kapitel 2.5 beschrieben.

Auf Grund dieser Komplexität des Schach Spiels, sowie der begrenzten Rechenkapazitäten der damaligen Computer, wurde dies auf eine Tiefe von vier begrenzt. Dies bedeutet, dass der Computer lediglich vier Züge vorausschaut. Zusätzlich schaut der Computer lediglich sieben verschiedene Optionen pro Zug an. Die Auswahl dieser sieben Optionen geschah mittels simpler heuristischer Berechnungen, die versuchten im Vorab die sieben aussichtsreichsten Optionen zu wählen. [?]

Diese Limitationen jedoch ermöglichten lediglich ein relativ simples, wenn auch passables Schachspiel.

Um die Zahl der zu evaluierenden Züge zu reduzieren, wurde im Jahr 1958 von Allen Newell und Herbert Simon eine Modifizierung des Minimax Ansatzes veröffentlicht - genannt “alpha beta pruning”. Diese Modifizierung verhindert das Evaluieren von Zügen, die eindeutig schlechter sind. Dieser Ansatz wird genauer in Kapitel 2.4 beschrieben. [?]

Diese neue Vorgehensweise kann einiges an Rechenkapazitäten sparen und soll so zum ersten Mal einen menschlichen Spieler geschlagen haben, der das Spiel allerdings erst kurz zuvor erlernt hatte. [?]

Durch Verbesserung der Algorithmen und erweiterten Rechenkapazitäten durch immer bessere und performantere Computer, teilweise extra optimiert auf das Schach Spiel,

konnten Stück für Stück neue Erfolge erzielt werden. 1962 konnte eine KI von Arthur Samuel erstmals einen renommierten Spieler, Robert Nealy, schlagen. [?]

1994 konnten die besten Dame Spieler der Welt eine KI namens “CHINOOK” nur noch ein Unentschieden abverlangen. [?] Bereits 1988 gelang einem Programm namens “Deep Thought” von Feng-hsiung Hsu, später von IBM weiterentwickelt unter dem Namen “Deep Blue” bekannt, ähnliches im Schach Spiel und so schlug die KI den Schach Weltmeister. [?]

Beide Programme basierten dabei auf drei wesentlichen Punkten: Eine Datenbank von Eröffnungszügen entnommen von professionellen Spielern, alpha-beta Suchbäume mit einer Menge an Evaluierungsfunktionen sowie einer Endspiel Datenbank sobald nur noch eine geringe Anzahl an Spielfiguren existiert. [?]

Diese Programme benötigten speziell optimierte Computer, die auf die nötigen Berechnungen für die jeweiligen Spiele ausgelegt waren. Anders handhabt es das Programm “Stockfish” aus dem Jahre 2008. Dies ist auf jedem Computer ausführbar und dennoch für einen Menschen scheinbar nicht schlagbar.

Lange war Stockfish unbesiegt, dies änderte sich jedoch im Jahr 2017 als AlphaZero mit 64:36 gegen Stockfish gewann. AlphaZero ist dabei ein verallgemeinerter Ansatz von AlphaGo Zero. Dies wiederum ist eine Aktualisierung von AlphaGo, das im Jahr 2016 den Weltmeister in Go schlug. Dabei verwendet AlphaGo neuronale Netzwerke und verwendet Methoden des “supervised learning” und “reinforcement learning”, um sich selbstständig weiter zu entwickeln. [?]

An diesem Punkt endet die Entwicklung von immer besser werdenden KIs jedoch nicht. Besonders durch den Fortschritt im Bereich der neuronalen Netze und speziell im “Deep Learning” Bereich werden immer komplexere Spiele durch den Computer beherrscht, immer öfter auch besser als vom Menschen. Ein Beispiel stellt dabei Alpha Star dar, das im Jahre 2019 einer der besten Teams im Echtzeit-Strategiespiel “Starcraft 2” geschlagen hat. Diese Entwicklung wird sich wohl auch in den kommenden Jahren fortsetzen, auch außerhalb der Spiele Szene, weshalb KI in unserer Gesellschaft immer mehr an Bedeutung gewinnt. [?]

## 2.3 Minimax-Algorithmus

Der Minimax Algorithmus basiert auf dem “Minimax-Theorem” von John Vonn Neumann.

Der Hauptsatz des Theorems für 2 Personen Spiele lautet:

In der gemischten Erweiterung  $(X, Y, G')$  eines jeden 2-Personen-Nullsummenspiels mit endlichen (reinen) Strategieräumen A und B existiert eine Konstante  $V$  und für jeden Spieler mindestens eine (gemischte) Gleichgewichtsstrategie  $x^*$  bzw.  $y^*$ , mit der er eine erwartete Auszahlung von mindestens  $V$  garantieren kann.

Für Spieler A existiert ein  $x^* = x_1^*, \dots, x_i^*, \dots, x_m^*$  mit  $x_i^* \geq 0$  und  $\sum_{i=1}^m x_i^* = 1$ ,  
so dass  $\max_x \min_y G'(x, y) = \min_y G'(x^*, y) = V$ .

Für Spieler B existiert ein  $y^* = \{y_1^*, \dots, y_j^*, \dots, y_n^*\}$  mit  $y_j^* \geq 0$  und  $\sum_{j=1}^n y_j^* = 1$ ,  
so dass  $\min_y \max_x G'(x, y) = \max_x G'(x, y^*) = V$ . [?]

Dabei wird das Spiel von Spieler A gestartet, der somit zuerst eine Strategie wählt. Die Annahme dabei ist, dass Spieler A davon ausgeht, dass Spieler B stets den für ihn besten Zug spielen wird. Das bedeutet, dass Spieler A die Strategie wählt, bei der unter dieser Annahme dennoch das beste Endergebnis zu erreichen ist. Hat Spieler A also eine Menge  $S$  an Strategien zur Auswahl, werden alle Szenarien von jedem Zug  $s \in S$  evaluiert. Die Bewertung dieser Strategien nennen wir  $u(s) | s \in S$ . Diese Evaluation erfolgt dabei aus Perspektive von Spieler A. Dies bedeutet, dass umso höher das Ergebnis von  $u(s)$  ist, desto besser ist das Ergebnis für Spieler A. Je niedriger das Ergebnis, desto besser für Spieler B.

Dann wird das Minimum aller Bewertungen der Endzustände für jede Strategie  $s$  gewählt. Diese Strategien nennen wir  $\min(s)$  und dessen Bewertung somit  $u(\min(s))$ , wobei gilt  $s \in S$ . Nun wird die Strategie  $s_{best} \in S$  gewählt, so dass gilt:

$$\forall s \in S : (s \neq s_{best} \rightarrow u(\min(s)) \leq u(\min(s_{best}))) \quad (2.1)$$

Anders ausgedrückt - existiert also eine Menge  $X$  von Strategien von Spieler A und eine auf  $X$  konvexe Menge  $Y$  von möglichen Strategien von Spieler B, so lautet die Optimierungsregel für Spieler A

$$\max_X [\min_Y u(X, Y)] \quad (2.2)$$

Umgekehrt versucht Spieler B das für Spieler A ungünstigste Ergebnis zu wählen und wählt daher die Strategie, die das Minimum der für Spieler A jeweils nach dem Zug noch bestmöglichen Ergebnisse versprechen. Somit lautet die Optimierungsregel für Spieler B

$$\min_Y [\max_X u(X, Y)] \quad (2.3)$$

Dadurch kann Spieler B den Ertrag des Spielers A auf diesen Wert begrenzen. Es gilt also

$$\max_X [\min_Y u(X, Y)] \leq \min_Y [\max_X u(X, Y)] \quad (2.4)$$

Das Theorem geht dabei davon aus, dass es also einen Sattelpunkt  $v$  geben muss, bei dem sich die beiden Optimierungen für Spieler A und B einpendeln. Dieser Sattelpunkt lautet

$$\max_X [\min_Y u(X, Y)] = \min_Y [\max_X u(X, Y)] = v \quad (2.5)$$

Diese Strategie basiert auf puren Berechnungen und ist für einen Computer somit leicht durchführbar. Dafür verlangt es folgende Informationen [?]:

- *States*: Alle möglichen Zustände des Spiels
- $s_0$ : Anfangszustand des zu betrachtenden Spiels
- $player(s)$ : Gibt den Spieler zurück, der in gegebenem Zustand am Zug ist
- $actions(s)$ : Eine Liste aller möglichen Züge ausgehend von Zustand  $s$
- $result\_states(s, a)$ : Der von Zustand  $s$  über Zug  $a$  erreichbare Zustand
- $terminal\_test(s)$ : Prüft einen Zustand darauf, ob dieser das Ende des Spiels bedeutet. Diese wird definiert als

$$terminal\_test : States \rightarrow \mathbb{B} \quad (2.6)$$

wobei  $\mathbb{B}$  einem Boolean-Wert, sprich wahr oder falsch, entspricht.

Mittels dieser Funktion kann eine Menge *terminalStates* gebildet als Menge aller Endzustände gebildet werden:

$$terminal\_states := s \in S | terminal\_test(s) \quad (2.7)$$

- $utility(s, p)$ : Bewertet einen Zustand, indem sie diesem einen numerischen Wert zuweist. Diese Funktion ist definiert durch

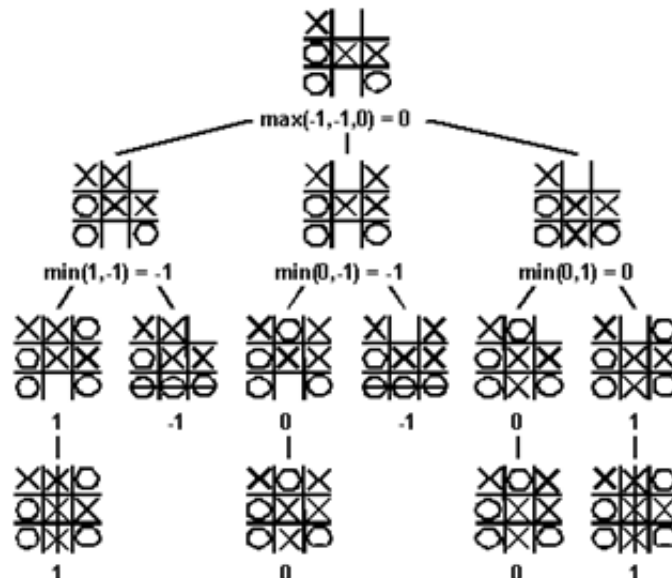
$$utility : terminal\_states \times player \rightarrow \mathbb{N} \quad (2.8)$$

wobei diese den Zustand aus Sicht des gegebenen Spielers bewertet. Umso höher die Zahl  $N$  also, desto besser das Ergebnis für Spieler  $P$ .

Um die erreichbaren Zustände zu erhalten, benötigt es einen Algorithmus, der anhand der Spielregeln und eines Ausgangszustands  $s_0$  und einer Liste aller Züge von  $s$   $actions(s)$  alle erreichbaren Zustände  $result_s tates$  berechnet. Von jedem Zustand  $s \in result_s tates$  wird dann wiederum jeder mögliche erreichbare Zustand berechnet. Diese Schleife wird fortgeführt, bis die berechneten Zustände den Status einen Endzustand erreichen. Der Fall ist dies, wenn  $terminal_{test}(s) = true$  ergibt und somit von diesem Zustand aus keine Änderungen mehr möglich sind.

Entscheidend für die Wahl eines Zuges ist dann die Bewertung jedes einzelnen Zustandes. Dies ist solange umsetzbar, wie der Computer ohne Probleme alle möglichen Zustände berechnen und evaluieren kann. Um dies an einem Beispiel zu zeigen - Beim Spiel "Tic Tac Toe", bei dem in einem 3x3 Feld zwei Spieler gegeneinander antreten mit dem Ziel drei aneinander angrenzende (vertikal, horizontal oder diagonal) Felder mit ihrer Figur zu belegen, gibt es insgesamt 255.168 verschiedene Spielverläufe. [?] Diese sind von heutigen Computern in akzeptabler Zeit berechnen- und auswertbar.

Dazu bildet der Computer einen sogenannten Minimax-Baum und wählt dann den erfolgversprechendsten Zug aus. Nach jedem getätigten Zug werden die erreichbaren Zustände nur noch vom neuen Zustand aus berechnet. Im Spätspiel kann ein solcher Baum bei "Tic Tac Toe" beispielsweise aussehen wie in Abbildung 2.1.



Tic Tac Toe Minimax Baum [?]

Dabei werden vom Ausgangszustand, der den Zustand des aktuellen Spiels widerspiegelt, aus alle möglichen Folgezustände berechnet. Von diesen werden wiederum alle möglichen

Folgezustände berechnet. Dies wird solange fortgeführt, bis alle neu berechneten Zustände Endzustände sind. Dann werden alle Endzustände evaluiert. Eine 1 stellt dabei einen Sieg dar, eine 0 ein Unentschieden und eine -1 eine Niederlage. Da davon ausgegangen wird, dass der gegnerische Spieler stets den besten Zug auswählt, wird jeder Zustand, der noch Folgezustände besitzt, mit dem für ihn schlechtesten Wert aller seiner möglichen Folgezustände bewertet. Der Computer wählt dann den Zustand mit der für ihn besten Bewertung. [?]

In diesem Beispiel wird sich der Computer somit für den dritten Zug von links entscheiden, da bei beiden anderen ein Sieg des Gegenspielers bevorsteht, sollte dieser jeweils die perfekten Züge spielen. Beim dritten Zug kann der Gegner maximal noch ein Unentschieden erreichen.

Diese Strategie gerät jedoch dann an ihre Grenzen, wenn nicht mehr alle möglichen Zustände berechnet werden können. Bei dem Spiel Schach beispielsweise belaufen sich Schätzungen schon nach den ersten 40 Spielzügen auf  $10^{115} - 10^{120}$  verschiedene Spielverläufe. [?] Dies ist auch für einen Computer in tolerierbarer Zeit unmöglich berechenbar. Aus diesem Grund muss die Strategie für solch komplexere Spiele abgewandelt werden. Die verschiedenen Ansätze dazu sind in Kapitel 2.5 beschrieben.

Um diesen Algorithmus in die Praxis umzusetzen, verlangt es drei Funktionen, die jeweils auf den Parameter **state** angewiesen sind. Dieser Parameter gibt Aufschluss über den aktuellen Zustand des Spiels.

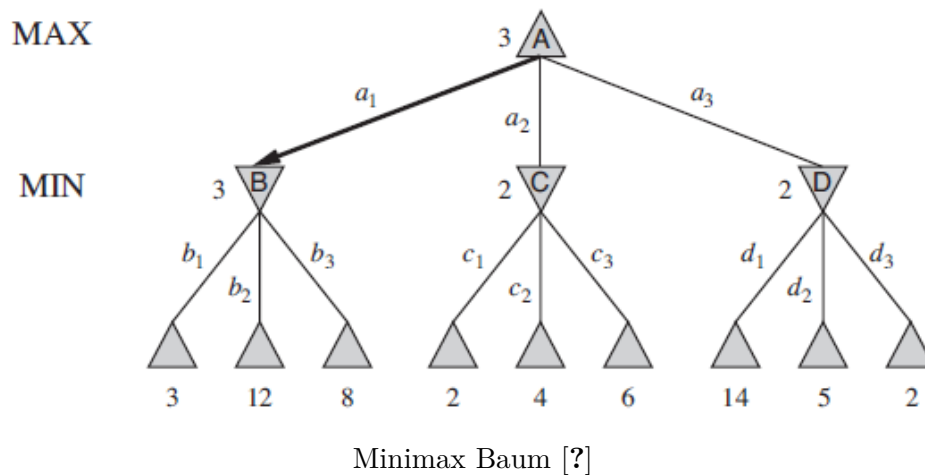
Zusätzlich verlangt der Algorithmus neben der genannten Funktionen *utility* und *finished* eine Funktion *min* sowie eine Funktion *max*, die jeweils mehrere Werte vergleichen und den Minimum bzw. Maximum aller verglichenen Werte zurück geben. Eine beispielhafte Implementierung eines solchen mittels drei verschiedener Funktionen kann in Algorithmus 2 gesehen werden.

Die erste Funktion geht dabei jeden möglichen Zug vom gegebenen Ursprungszustand durch und gibt am Ende den Zustand zurück, der den besten Minimum-Wert durch die Evaluierungsfunktion *utility* erreicht. Dazu gibt die Funktion jeden der erreichbaren Zustände zu der *minValue* Funktion. Diese gibt entweder den durch die *utility* Funktion errechneten Wert des Zustands zurück, sollte der Zustand ein Endzustand sein, oder sie gibt das Minimum aller Maxima-Werte der erreichbaren Zustände zurück. Dazu wiederum wird die *maxValue* Funktion verwendet, die genau das Gegenteil der *minValue* Funktion macht. Ist der Endzustand erreicht, gibt zwar auch die *maxValue* Funktion den Wert des Zustands direkt zurück, andernfalls aber gibt sie das Maximum aller Minimum Werte der vom gegebenen Zustand aus erreichbaren Zustände zurück, wozu wiederum auf die



minValue Funktion zurückgegriffen wird. Dies ganze wiederholt sich also rekursiv so lange, bis alle neu errechneten Zustände den Wert eines Endzustandes erreicht haben. [?]

Diesen Endzuständen werden dann Werte zugewiesen, die aufsteigend rekursiv miteinander verglichen und abwechselnd Minimum und Maximum gewählt werden, um dem Minimax-Theorem dahingehend zu folgen. Um das Ganze zu verdeutlichen kann Abbildung 2.2 als Beispiel genommen werden:



Dabei hat der gegebene Zustand noch eine erreichbare Tiefe von drei bis alle darauffolgenden Zustände Endzustände sind. Diesen Endzuständen wird dann ein Wert zugewiesen. Nun werden diese Werte auf einer um eins höheren Ebene verglichen. Da es sich hierbei um eine gerade Tiefe (2) handelt, wird hier das Minimum dieser Werte gewählt. Im Knoten B ist dies in unserem Beispiel 3, das 3 der geringste Wert der Endzustände (3, 12 und 8) ist. Im Knoten B sowie D ist jeweils 2 der geringste mögliche Wert.

Von diesen ausgerechneten Minima wird dann auf Ebene 1 das Maximum berechnet. Das Maximum von 3, 2 und 2 beträgt 3, weshalb der Wert von Knoten B der höchste ist und somit wird der Zug, der zu Zustand B führt, zurückgegeben.

Somit ist es mit dem Minimax-Algorithmus möglich anhand für den Computer möglicher Berechnungen ein Spiel zu evaluieren und den besten Zug zu wählen unter der Annahme, dass der Gegenspieler ebenso handeln wird. Problematisch dabei ist jedoch besonders bei komplexeren Spielen die Dauer der Evaluierung aller Züge. Dies liegt zum einen daran, dass jeder einzelne Zug evaluiert wird und somit bei komplexen Spielen eine enorm hohe Zahl an Zuständen evaluiert werden muss. Zum anderen ist das größte Problem aber wohl, dass es zwangsweise das Spiel bis in die Endzustände berechnen muss, was eine enorm hohe Tiefe verlangt und somit eine extreme Zeitspanne zum Berechnen mit sich zieht. So ist die Zeitkomplexität bei einem Spiel der Tiefe  $m$  und einer Anzahl  $b$  an möglichen Zügen gleich  $O(b^m)$ . [?]

Das erste Problem, dass jeder einzelne Zug berechnet wird, wird mit dem Ansatz des Alpha Beta Pruning versucht zu minimieren. Dieser Ansatz wird im Kapitel 2.4 erklärt. Das Problem, dass stets bis in die Endzustände gerechnet werden muss, wird in Kapitel 2.5 genauer erläutert sowie einige Lösungsansätze dargestellt.

## 2.4 Alpha-beta pruning

Ein Problem bei dem Minimax Algorithmus ist, wie bereits in Kapitel 2.3 angesprochen, die Evaluierung eines jeden möglichen Zuges, obwohl manche Züge eventuell schon im Vorhinein mittels trivialer Berechnungen ausgeschlossen werden können und gar nicht mehr näher betrachtet werden müssten. Auch ein guter Schachspieler geht ja nicht jeden möglichen Zug im Kopf durch, sondern schaut sich nur spezielle Züge bis zu einer gewissen Tiefe an und sobald er erkennt, dass dieser nicht gewinnbringend ist, schließt er diesen direkt aus, ohne ihn weiter zu evaluieren.

Für ein ähnliches Schema gibt es eine erweiternde Technik des Minimax-Algorithmus. Diese nennt sich Alpha-Beta-Pruning. Dabei wird versucht große Teile des Minimax Baums bereits auszuschließen, bei denen auf triviale Weise erkannt werden kann, dass diese die finale Entscheidung nicht beeinflussen würden.

Durch die Verzweigung von min- und max-Funktionen im Minimax-Baum, können bestimmte Zweige oftmals bereits ausgeschlossen werden, da dieser für das Ergebnis der min-max Verzweigung irrelevant ist. [?] Um dies an einem Beispiel zu erklären, kann folgende min-max-Verschachtelung dienen:

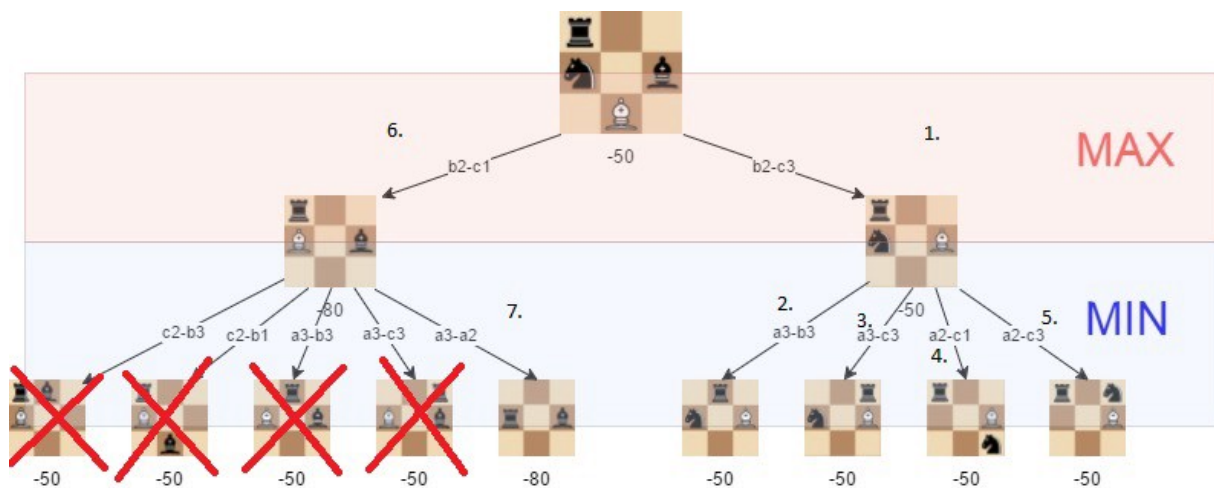
$$MINIMAX(root) = \max(\min(3, 12, 8), \min(2, x, y)) \quad (2.9)$$

Aus dem ersten min Zweig wird dabei 3 als Gewinner hervorgehen, da es der niedrigste Wert ist. Im zweiten Zweig gibt es mit dem Wert 2 bereits einen potentiellen Gewinner. Auch bei weiterer Evaluierung dieses Zweigs kann höchstens eine noch kleinere Zahl als 2 als Sieger aus dem min Zweig hervorgehen. Da 2 oder eine kleinere Zahl in dem darüberstehenden max-Zweig sich nicht gegen 3 durchsetzen kann, ist die Evaluierung der bisher nicht berechneten Zweige x und y irrelevant, da sie keinen Einfluss auf die Entscheidung haben. Dementsprechend kann diese übersprungen werden. Der Baum sieht dann wie folgt aus

$$\begin{aligned}
 MINIMAX(root) &= \max(\min(3, 12, 8), \min(2, x, y)) \\
 &= \max(3, \min(2, x, y)) \\
 &= \max(3, z) \quad (2.10) \\
 &= 3 \quad (\text{where } z = \min(2, x, y) \leq 2)
 \end{aligned}$$

Dabei wird der Term  $\min(2, x, y)$  durch  $z$  ersetzt, wobei  $z$  durch die Eigenschaft  $z \leq 2$  definiert wird. Auf Grund dieser Tatsache, kann das Ergebnis des Terms  $\max(3, z)$  eindeutig auf 3 festgelegt werden.

In Abbildung 2.3 kann dies nochmal anhand eines realen Beispiels von einem Ausschnitt aus einem Schachspiel verdeutlicht werden.



Alpha Beta Pruning Beispiel anhand von Schachausschnitt [?]

Gehen wir dabei aus, dass zuerst der rechte Zweig (1.) vollständig evaluiert wurde, wofür zunächst alle Knoten dieses Zweigs (2.-5.) betrachtet wurden. Nach diesen Berechnungen wird Knoten 1 das Minimum aller Zweige von 1 zugewiesen. Dieses beträgt -50. Danach soll der linke Zweig (6.) evaluiert werden. Dabei wird zuerst Knoten 7 betrachtet und ein Ergebnis für -80 ermittelt. Da Knoten 6 das Minimum aller seiner Zweige ermitteln wird, wird das Ergebnis kleiner oder gleich -80 sein. Der Ursprungsknoten wird später also das Maximum aus -50 und  $z | z \leq -80$  ermitteln, das unabhängig vom endgültigen Wert für  $z$  auf jeden Fall -50 betragen wird. Deshalb müssen alle anderen Zweige von 6 gar nicht mehr betrachtet werden, da diese ohnehin kein Einfluss auf das Ergebnis hätten.

Diese Methode führt zu einer erheblichen Einsparung an zu evaluierenden Positionen und erhöht die Performanz des Minimax Algorithmus dadurch deutlich. Im Beispiel zu sehen in Abbildung 2.4 hätten wir bei einer Tiefe von 4 mit einem herkömmlichen Minimax Algorithmus noch ganze 879.750 Positionen zu evaluieren. Durch die Verbesserung des

Algorithmus mittels Alpha-Beta-Pruning sinkt diese Zahl der zu evaluierenden Positionen um ein beachtliches auf lediglich noch 61.721, und somit um mehr als ein zehnfaches. [?]

8								
7								
6								
5								
4								
3								
2								
1								
	a	b	c	d	e	f	g	h

	Minimax	Minimax with alpha-beta
Positions	879750	61721

Beispielhafte Schachpositionierung [?]

Zur Umsetzung eines solchen Algorithmuses werden zwei Werte benötigt -  $\alpha$  und  $\beta$ :

$\alpha$  gleicht dabei dem bisher besten (sprich höchsten) Wert in dem max Pfad des Minimax-Baums.

$\beta$  gleicht dagegen dem bisher besten (sprich niedrigsten) Wert in dem min Pfad des Minimax-Baums. [?]

Diese Werte werden kontinuierlich aktualisiert und sobald bekannt ist, dass der Wert eines Knoten schlechter als das aktuelle  $\alpha$  (bei max Ebenen) beziehungsweise  $\beta$  (bei min Ebenen) sein wird, werden die restlichen Zweige dieses Knoten nicht mehr in Betracht gezogen und somit übersprungen. [?]

Dies kann in einen Algorithmus verpackt werden, wie in Algorithmus 3 zu sehen.

Dieser Algorithmus gleicht im Grunde dem Minimax Algorithmus, vorgestellt in Kapitel 2.3. Jedoch unterscheiden die beiden Algorithmen sich um die ergänzten Werte  $\alpha$  und  $\beta$ . Diese werden dabei jeweils in die Funktionen maxValue bzw. minValue mitgegeben. Innerhalb dieser wird dann beim Iterieren über alle erreichbaren Zustände geprüft, ob einer dieser Zustände bereits größer als  $\beta$ , das bisherige Minimum im min Pfad, im Fall der maxValue Funktion beziehungsweise kleiner als  $\alpha$ , das bisherige Maximum im max-Pfad, im Fall der

minValue Funktion ist. In dem Fall wird direkt der errechnete Wert zurückgegeben und Evaluierungen aller weiteren Zustände ausgelassen, da diese ohnehin irrelevant wären.

Die Effektivität dieses Algorithmus hängt jedoch von der Reihenfolge der Evaluierung der einzelnen Zustände ab. Im besten Fall werden stets zuerst die scheinbar besten Zustände evaluiert, die ein weiteres Evaluieren der restlichen Zustände weitestgehend überflüssig machen. Im besten Fall könnte so die in Kapitel 2.3 angesprochene Zeitkomplexität eines Spiels mit einer Tiefe  $m$  und einer Anzahl  $b$  an möglichen Zügen von  $O(b^m)$  auf  $O(b^{\frac{m}{2}})$  reduziert werden. Bei einer zufälligen Auswahl der Reihenfolge von zu evaluierenden Zügen dagegen würde sich die Zeitkomplexität immerhin noch auf  $O(b^{\frac{3m}{4}})$  reduzieren. [?]

Verschiedene Ansätze können dabei helfen die Reihenfolge der zu evaluierenden Züge zu verbessern. Einer dieser Ansätze ist die Züge zu sortieren, so dass zunächst die Züge evaluiert werden, bei der Figuren geschlagen werden können und daraufhin die, bei denen Bedrohungen aufgebaut werden können. Von den übrig gebliebenen Zügen werden dann zunächst die Vorwärtsbewegungen evaluiert und erst zum Schluss die Rückwärtsbewegungen. Nach diesem Schema kann die Zeitkomplexität ungefähr auf  $2 * O(b^{\frac{m}{2}})$  gebracht werden. [?]

Andere Methoden sind die Züge zusätzlich danach zu sortieren, welche Züge bei vergangenen Spielen zu Erfolg geführt haben. Eine Methode dazu ist, zunächst die Züge der ersten Ebene zuerst zu evaluieren und dann eine Ebene tiefer zu gehen, die Züge dabei aber nach den Ergebnissen aus der ersten Evaluierungsiteration zu sortieren. Dies nennt sich "Iterative Deepening". [?]

Diese Optimierung des Minimax Algorithmus spart bereits einige zu evaluierende Zustände und somit einiges an Rechenkapazitäten. Die Anzahl aller möglichen Spielzüge in Schach bleibt jedoch zu hoch, um über jeden einzelnen in angemessener Zeit zu iterieren. Auf Grund dessen muss die Tiefe, nach der die Züge evaluiert werden, reduziert werden. Ansätze dazu werden in Kapitel 2.5 beschrieben.

## 2.5 Problematik komplexerer Spiele

Der in Kapitel 2.3 beschriebene Minimax-Algorithmus hat - auch mit der Optimierung durch Alpha-Beta-Pruning (Kapitel 2.4) - das Problem, dass zur Entscheidungsfindung alle Pfade bis zum Ende des Spiels abgegangen werden müssen. Beim Spiel Schach beispielsweise gibt es schon nach 40 Spielzügen  $10^{120}$  Möglichkeiten das Spiel zu führen. Selbst unter der sehr optimistischen Annahme, dass eine Million verschiedene Züge pro Sekunde evaluiert werden könnten, würde die Evaluation aller Spielzüge  $10^{108}$  Jahre benötigen. [?] Zum Vergleich - das Alter der Erde wird auf nicht viel mehr als  $10^9$  Jahre geschätzt. [?] An

dieser Tatsache ändert auch eine Reduzierung der zu evaluierenden Möglichkeiten durch Alpha-Beta Pruning nichts Wahrnehmbares.

Die Lösung dieses Problems scheint auf der Hand zu liegen - die Reduzierung der zu evaluierenden Züge, indem die einzelnen Pfade nicht bis zum Ende des Spiels abgegangen werden, sondern nur bis zu einer bestimmten Tiefe.

Das erste Problem bei diesem Ansatz jedoch ist, dass Spiele nicht mehr nach dem Kriterium "Sieg", "Niederlage" und "Unentschieden" bewertet werden können, da dies zu dem gegebenen Zeitpunkt unter Umständen noch nicht absehbar ist. Deshalb sind bei dieser Methode andere Funktionen zur Evaluierung der einzelnen Zustände nötig. Die verschiedenen Möglichkeiten zur Evaluierung der Zustände des Schachspiels werden in Kapitel 2.6 näher beschrieben.

Zum Abbrechen der redundanten Aufrufe der Funktionen bei einer gewissen Bedingung, beispielsweise einer festgelegten Tiefe, muss diese in die Abbruchbedingung der rekursiven Funktion(en) hinzugefügt werden.

Im simpelsten Ansatz einer festgelegten Tiefe, die die einzelnen Pfade maximal durchlaufen werden, kann dies mittels einer Funktion "cutoff\_test" gemacht werden, die die Parameter des aktuellen Spielzustands sowie die Tiefe übermittelt bekommt. Sie gibt den Wert wahr zurück, wenn der mitgegebene Zustand ein Endzustand oder wenn die Tiefe größer oder gleich der festgelegten Tiefe ist. [?]

Wichtig ist dabei, dass bei jedem rekursiven Aufruf der Funktion der festgelegte Tiefen-Parameter stets um eins steigt, damit dieser Wert für die Abbruchfunktion zuverlässig verwendet werden kann. Als Beispiel ist hier eine modifizierte Version der maxValue Funktion aus Algorithmus 3 zu sehen.

Ein weiterer Ansatz an Stelle einer festgelegten Tiefe ist, eine Zeit festzulegen, nach der die Rekursion abgebrochen wird. Dies hat den Vorteil, dass sich die Tiefe von selbst anpasst, je nachdem wie komplex das Spiel ist. Im Anfang des Spiels, wenn noch viele verschiedene Züge möglich sind, ist es in akzeptabler Zeit nicht möglich, sehr weit in die Tiefe zu schauen. Wenn jedoch gegen Ende des Spiels nur noch wenige Züge durchführbar sind, ist es von großem Vorteil, diese in höherem Detail zu begutachten, um den besten Zug auswählen zu können. Bei einer fixen Tiefe müsste sich auf einen Kompromiss der Tiefe geeinigt werden, um zugleich eine akzeptable Berechnungszeit zu Anfang des Spiels und eine möglichst detaillierte Betrachtung gegen Ende zu ermöglichen.

Um diese zeitbasierte Abbruchbedingung zu ermöglichen, wird "Iterative Deepening" verwendet. Dabei wird die Tiefe stückweise erhöht. Zunächst werden Züge also lediglich mit einer Tiefe von 1 betrachtet. Ist danach noch Zeit übrig wird die Rekursion um eine

---

**Algorithm 1** Tiefenlimit Alpha Beta Pruning [?]
 

---

```

function max_value(state,  $\alpha$ ,  $\beta$ , depth)
  if terminal_test(state, depth) then
    return utility(state)
  end if
   $v \leftarrow -\infty$ 
  for all ainactions(state) do
     $v \leftarrow \max(v, \min\_value(\text{result\_state}(\text{state}, a), \alpha, \beta, \text{depth} + 1))$ 
    if  $v \geq \beta$  then
      return  $v$ 
    end if
     $\alpha \leftarrow \max(\alpha, v)$ 
  end for
  return  $v$ 
end function

function cutoff_test(state, depth)
  if terminal_test(state) then
    return True
  end if
  if depth  $\geq MAX\_DEPTH$  then
    return True
  end if
  return False
end function

```

---

Stufe tiefer durchgeführt. Dies wiederholt sich so lange bis das Zeitlimit abgelaufen ist. [?]

Dieses stückweise Vertiefen hilft zusätzlich dabei die zu betrachtenden Züge zu sortieren, um das Alpha-Beta-Pruning zu optimieren, wie in Kapitel 2.4 beschrieben. Eine Gefahr bei diesen Methoden, die das Spiel nicht ganz betrachten, ist, dass die Evaluierungsfunktionen meist nur Aufschluss über den aktuellen Zustand geben, aber nicht potentielle Gefahren für die Zukunft betrachten. Gewinnt ein Spieler beispielsweise mit einem Zug einen Bauern erscheint das einer simplen, heuristischen Berechnung des Zustandes zunächst als Gewinn. Stellt der Spieler dabei jedoch beispielsweise seine Dame in eine Position, in der diese geschlagen werden kann, würde er im nächsten Zug sehr wahrscheinlich eine Figur von wesentlich höherem Wert verlieren. Eine Möglichkeit dabei ist nur ruhende Zustände zu betrachten. Diese Methode nennt sich “quiescence search”. Dabei werden alle Züge des zu betrachtenden Zustands, die ein Schlagen einer Figur zur Folge haben, durchgespielt. Ist keine Figur mehr zu schlagen, sondern lediglich noch Bewegungszüge möglich, so wird der Zustand als ruhig bewertet und dieser mittels der entsprechenden Funktionen evaluiert. [?]

Eine andere Möglichkeit ist, die Evaluierungsfunktion dahingehend anzupassen, dass nicht nur der nächste Zustand heuristisch berechnet wird, sondern beispielsweise auch attackierte Figuren oder Schwachstellen in der Verteidigung in die Evaluation mit einzubeziehen. Darauf und generell auf möglich Evaluierungsfunktionen wird näher in dem hier folgenden Kapitel 2.6 eingegangen.

## 2.6 Evaluierungsfunktionen

Entscheidend für eine Schach-spielende-KI ist - neben der Anzahl der zu betrachtenden Spielzüge - die Art der Evaluation, die Aufschluss darüber geben soll, wie aussichtsreich eine bestimmte Position ist. In diesem Kapitel werden verschiedene Ansätze aufgeführt und erklärt. Dabei wird zunächst auf die Spieleröffnung eingegangen, der eine Sonderrolle im Schachspiel zukommt. Danach erfolgen die beispielhafte Nennung verschiedener Evaluierungsfunktionen im Mittelspiel ehe zum Schluss auf Strategien für das Endspiel eingegangen wird.

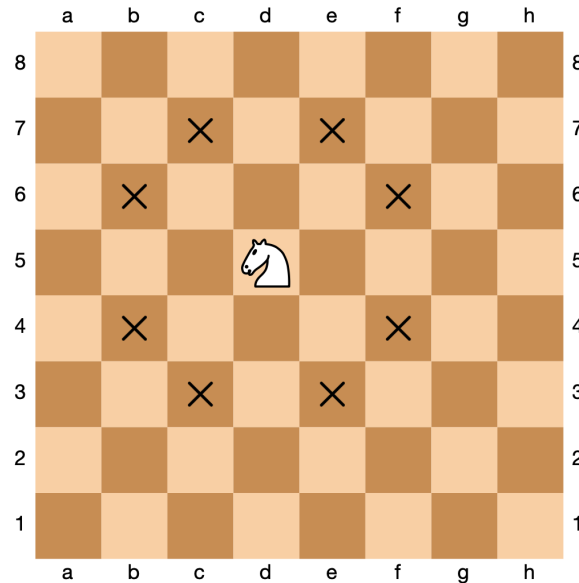
### 2.6.1 Eröffnungsstrategie

Wie bereits in den einführenden Abschnitten des zweiten Kapitels erläutert, gibt es viele verschiedene Züge um ein Schachspiel durchzuführen. Damit der Spieler mit einer möglichst guten Ausgangssituation in das Spiel startet sollten besonders die ersten Züge mit Bedacht gewählt werden. Hierbei ist es zielführend, wenn der Spieler seine Figuren direkt zu den vielversprechendsten Feldern zieht und somit in möglichst wenigen Iterationen zur gewollten Ausgangssituation kommt.[?]

Ein Beispiel für eine gute Ausgangssituation ist es, die vier zentralen Felder d4, e4, d5, e5 zu kontrollieren. Diese Felder haben die Eigenschaft, dass ihre Lage jeder Schachfigur die jeweils größtmögliche Reichweite ermöglicht. Somit ist dies für jede Figur mitunter die gefährlichste Position im Spiel. Zum Beispiel kontrolliert ein Springer auf dem Feld d4 acht Felder, wohingegen ein Springer auf dem Feld b1 oder g1 nur zwei Felder kontrolliert (siehe Abbildung 2.5). Ein solches explizites Vorgehen bei der Eröffnung eines Spiels nach einer speziellen Strategie, wird als Eröffnungsstrategie bezeichnet.[?]

Durch die Verfügbarkeit von Minimax-, oder der Alpha-Beta-Suche, stellt sich die Frage, ob solche im Vorhinein festgelegten Eröffnungsstrategien nicht überflüssig sind und durch die Suche des optimalen Zugs, unter der Beachtung der vermeintlichen nächsten n-Züge, das beste Ergebnis zu Stande kommt.





Beispielhafte Reichweite eines Springers auf dem Feld d5

Wie bereits Stuart Russell und Peter Norvig in *Artificial Intelligence – A Modern Approach* beschreiben, ist das Verwenden von solchen Algorithmen während der Eröffnung eines Schachspiels zu überdimensioniert und gleicht dem metaphorischen „Mit Kanonen auf Spatzen schießen“. Denn die Suche eines passenden Eröffnungszuges aus einer Anzahl von etwa einer Milliarden Spielständen ist sehr zeitaufwendig und überflüssig, wenn bspw. dies der erste Zug des kompletten Spiels ist.[?] Ebenfalls ist die Wahrscheinlichkeit, dass die Eröffnungszüge eines Spiels bereits bekannt und dokumentiert sind höher, als im späteren Spielverlauf, in dem sehr viele verschiedene mögliche Züge zur Auswahl stehen.

Die Lösung zur Implementierung eines Eröffnungsspiels, ohne die Verwendung von komplexen Suchverfahren, liegt in den sogenannten Opening-Books. Opening-Books sind eine Ansammlung von verschiedenen dokumentierten Eröffnungsstrategien, die es ermöglichen auf eine aktuelle Spielsituation, in der Eröffnung, den passenden Zug zu finden.

Der Vorteil von Opening-Books im Gegensatz zu Suchverfahren ist, dass sie nur eine bestimmte Tiefe von Zügen unterstützen und somit deutlich schneller in der Berechnung der besten Züge sind. Ebenfalls bilden die Opening-Books sinnvolle Strategien ab, die teilweise bereits seit mehreren Jahrhunderten analysiert und verbessert wurden. Beispielsweise gibt es die Strategie „Ruy Lopez“, oder auch die „Spanische Eröffnung“ genannt, die bereits 1561 im Zuge der veröffentlich des Buchs "Libro de la invención liberal y arte del juego del ajedrez" von Ruy López de Segura publiziert wurde. Somit müssen nicht erst Berechnungen durchgeführt werden, die bemessen, ob bzw. inwiefern ein Zug in dieser Situation zielführend ist oder nicht. Stattdessen wird auf eine Datenbank von Zügen und deren Bemessungen zurückgegriffen.[?]

Sobald eine Strategie abgeschlossen ist oder das Opening-Book keinen sinnvollen Zug mehr auf eine komplexe Spielsituation hat, kommen die bereits eingeführten Suchverfahren zum Einsatz.

Der Nachteil der Opening-Books ist, dass diese nicht für jeden im Eröffnungsspiel getätigten Zug des Gegners eine passende Strategie besitzen. Die Opening-Books gehen davon aus, dass der Gegner ein Spieler ist, der auch professionelle Eröffnungsstrategien spielen wird und somit möglichst schnell die besten Felder mit seinen Figuren zu kontrollieren. Wenn es dazu kommt, dass ein Spieler durch Unwissenheit, oder bewusst nicht die vermeintlich besten Züge zur Eröffnung spielt, sondern Züge, die nicht im Opening-Book vorkommen, dann gibt es keine Strategie zu der aktuellen Spielsituation und das Opening-Book kann nicht mehr eingesetzt werden.

Damit das Schachspiel trotz der Tatsache, dass das Opening-Book keine sinnvolle Strategie liefern kann, fortgesetzt werden kann, muss Gebrauch von Suchverfahren gemacht werden. Hierbei ist zu beachten, dass dies, im Gegensatz zu Opening-Books, negative Auswirkungen, auf die im Eröffnungsspiel benötigte Zeit zur Berechnung von Zügen hat.

## 2.6.2 Figurenbewertung (+ Position)

Der wohl simpelste und gängigste, aber auch wichtigste Ansatz zur Evaluierung eines Schachbretts ist die heuristische Bewertung der auf dem Brett vorzufindenden Materialien. Dies bedeutet, dass die Figuren beider Spieler gezählt und voneinander subtrahiert werden. Das Ergebnis dieser Rechnung gibt Aufschluss darüber, welcher Spieler sich momentan in der aussichtsreicheren Position befindet.

Da jedoch nicht alle Figuren vom gleichen Wert sind, ist es vorteilhaft den einzelnen Figuren einen Faktor zuzuweisen, der Aufschluss über diesen gibt. Dies nähert die Funktion der Wertschätzung des Schachbretts an den tatsächlichen Wert an, da zwei Türme beispielsweise wichtiger als zwei Bauern sind. Im Ansatz ohne Faktorisierung der verschiedenen Spielfiguren würde dies keine Beachtung finden.

Claude Shannon hat dazu 1949 den einzelnen Figuren folgende Werte zugewiesen, die seitdem als Richtwerte für Evaluierungen gelten. [?]

Bauer	1
Springer	3
Läufer	3
Turm	5
Dame	9
König	200

Die Berechnung des Wertes eines Schachbretts kann dann wie folgt aussehen:

$$f() = 200 * (K_w - K_b) + 9 * (Q_w - Q_b) + 5 * (R_w - R_b) + 3 * (B_w - B_b) + 3 * (N_w - N_b) + 1 * (P_w - P_b)[?] \quad (2.11)$$

wobei K für König steht, Q für Dame, R für Turm, B für Läufer, N für Springer und P für Bauer. Die untergestellten Zeichen w und b stehen für weiß und schwarz.

Diese Berechnung gibt bereits einen recht guten Aufschluss über den Wert des Spiels. Noch besser wird diese heuristische Berechnung jedoch, wenn dabei auch die Positionen dieser Figuren mit einbezogen werden. Dazu hat Tomasz Michniewski folgende Matrizen vorgeschlagen [?]

Bauer								Springer							
0	0	0	0	0	0	0	0	-50	-40	-30	-30	-30	-30	-40	-50
50	50	50	50	50	50	50	50	-40	-20	0	0	0	0	-20	-40
10	10	20	30	30	20	10	10	-30	0	10	15	15	10	0	-30
5	5	10	25	25	10	5	5	-30	5	15	20	20	15	5	-30
0	0	0	20	20	0	0	0	-30	0	15	20	20	15	0	-30
5	-5	-10	0	0	-10	-5	5	-30	5	10	15	15	10	5	-30
5	10	10	-20	-20	10	10	5	-40	-20	0	5	5	0	-20	-40
0	0	0	0	0	0	0	0	-50	-40	-30	-30	-30	-30	-40	-50
Läufer								König							
-20	-10	-10	-10	-10	-10	-10	-20	-30	-40	-40	-50	-50	-40	-40	-30
-10	0	0	0	0	0	0	-10	-30	-40	-40	-50	-50	-40	-40	-30
-10	0	5	10	10	5	0	-10	-30	-40	-40	-50	-50	-40	-40	-30
-10	5	5	10	10	5	5	-10	-30	-40	-40	-50	-50	-40	-40	-30
-10	0	10	10	10	10	0	-10	-20	-30	-30	-40	-40	-30	-30	-20
-10	10	10	10	10	10	10	-10	-10	-20	-20	-20	-20	-20	-20	-10
-10	5	0	0	0	0	5	-10	20	20	0	0	0	0	20	20
-20	-10	-10	-10	-10	-10	-10	-20	20	30	10	0	0	10	30	20
Dame								Turm							
-20	-10	-10	-5	-5	-10	-10	-20	0	0	0	0	0	0	0	0
-10	0	0	0	0	0	0	-10	5	10	10	10	10	10	10	5
-10	0	5	5	5	5	0	-10	-5	0	0	0	0	0	0	-5
-5	0	5	5	5	5	0	-5	-5	0	0	0	0	0	0	-5
0	0	5	5	5	5	0	-5	-5	0	0	0	0	0	0	-5
-10	5	5	5	5	5	0	-10	-5	0	0	0	0	0	0	-5
-10	0	5	0	0	0	0	-10	-5	0	0	0	0	0	0	-5
-20	-10	-10	-5	-5	-10	-10	-20	0	0	0	5	5	0	0	0

Diese Matrizen können ergänzend zur Berechnung des Spielwertes genommen werden, um die Positionen der einzelnen Schachfiguren mit einbeziehen zu können. Dazu wird für alle Figuren der Wert aus der entsprechenden Position genommen und diese Werte zusammenaddiert.

Mittels Kombination aus diesen heuristischen Messmethoden kann bereits eine aufschlussreiche Zahl berechnet werden, wie aussichtsreich das aktuelle Spiel für den jeweiligen Spieler ist. Verstärkt werden kann dies jedoch durch weitere Funktionen, die auch Angriff, Verteidigung und Beweglichkeit des Spiels mit einbeziehen.

### 2.6.3 Verteidigung

Um die Stärke der Verteidigung eines Spielers zu messen, gilt vor Allem die Verteidigung des Königs als Messwert dafür. Dabei gibt es verschiedene Strategien.

Eine dabei ist die Position der Bauern zu messen, um herauszufinden, inwiefern das sogenannte "Bauern Schild" intakt ist. Ein Bauern Schild existiert dann, wenn der König durch vorstehende Bauern geschützt ist und keine Figur den König bedrohen kann, ohne vorher an diesem vorbei zu kommen. Eine Lücke in einem solchen Schild sollte dabei zu einer negativen Bewertung des Spielzustands führen. Dies kann noch dahingehend erweitert werden, dass eine Lücke in dem Bauern Schild umso höher in die Bewertung mit einfließt, desto mehr Material, also Figuren multipliziert mit ihren Werten, der Gegner noch auf dem Schachbrett hat. Dies führt dazu, dass die KI in dem Fall einer Lücke eher zu einem Figurentausch neigt, um Gefahren für den König zu eliminieren, wenn es an seiner eigenen Verteidigung mangelt. [?]

Ein weiterer Ansatz ist den Angriff auf die Königs-Zone zu messen und entsprechend zu bewerten. Die Königszone wird dabei definiert auf die um den König umliegenden Felder plus zwei bis drei weitere Felder voraus in Richtung des Gegenspielers. In Abbildung 2.6 werden diese Felder rot markiert.

Dann wird die Zahl der Figuren des Gegners gemessen, die diese Felder angreifen können. Dies wird in einer Variable "attacking\_pieces\_counter" gespeichert. Anhand dieser Zahl wird das Gewicht "attack\_weight" bestimmt. Dies kann folgender Tabelle entnommen werden: [?]



Königszone im Schachspiel vom Ausgangszustand []

attacking_pieces_counter	attack_weight
1	0
2	50
3	75
4	88
5	94
6	97
7	99

Zusätzlich wird der Wert der Angriffe berechnet. Dazu wird für jede Figur, die die Königszone angreifen kann, gezählt, wie viele Felder innerhalb dieser die Figur erreichen kann. Dies wird dann mit einer Konstante multipliziert abhängig vom Typ der Figur - 20 für einen Springer, 20 für einen Läufer, 40 für einen Turm und 80 für eine Dame. Für jede angreifende Figur wird diese Berechnung durchgeführt und das Ergebnis zur Variable "value\_of\_attacks" hinzugefügt. [?]

Abschließend werden diese berechneten Werte “attack\_weight” und “value\_of\_attacks” miteinander multipliziert und das Ergebnis durch 100 dividiert, wie in folgender Gleichung zu sehen: [?]

$$(value\_of\_attacks * attack\_weight[attacking\_pieces\_counter]) / 100 \quad (2.12)$$

Dieser Wert gibt dann guten Aufschluss darüber, wie gut oder schlecht die Sicherheit des Königs gewährleistet wird. Dabei werden jedoch nur Offiziersfiguren betrachtet.

Ein alternativer Ansatz dazu ist die Anzahl der attackierten Felder außer Acht zu lassen und stattdessen alle Figuren mit in die Berechnungen mit einzubeziehen. Dieser Ansatz wird beispielsweise von der Schach KI Stockfish genutzt. Dabei erhalten die verschiedenen Figuren verschiedene Werte für den Angriff zugeordnet. Ein Turm erhält dabei 3 Punkte, eine Dame 5. Alle anderen Figuren erhalten 2 Punkte. Zusätzlich können noch weitere Faktoren einbezogen werden, die im Königsangriff eine Rolle spielen. Dann werden alle Punkte zusammengezählt und anhand folgender Liste der Wert des Angriffs bestimmt. [?]

```
1 static const int SafetyTable[100] = {
2 0, 0, 1, 2, 3, 5, 7, 9, 12, 15,
3 18, 22, 26, 30, 35, 39, 44, 50, 56, 62,
4 68, 75, 82, 85, 89, 97, 105, 113, 122, 131,
5 140, 150, 169, 180, 191, 202, 213, 225, 237, 248,
6 260, 272, 283, 295, 307, 319, 330, 342, 354, 366,
7 377, 389, 401, 412, 424, 436, 448, 459, 471, 483,
8 494, 500, 500, 500, 500, 500, 500, 500, 500, 500,
9 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
10 500, 500, 500, 500, 500, 500, 500, 500, 500, 500,
11 500, 500, 500, 500, 500, 500, 500, 500, 500, 500
12 };
```

Mittels dieser Ansätze ist es möglich die Sicherheit seines eigenen Königs zu berechnen. Dies ist ein besonders wichtiger Ansatz bei der Evaluation eines Spielzustandes, damit die KI den eigenen König nicht unbedacht offen stellt, was langfristige, negative Folgen für sein Spiel haben kann.

## 2.6.4 Angriff

Neben der Verteidigung spielen auch die Möglichkeiten zum Angriff auf den Gegner eine Rolle, um einen Spielzustand zu evaluieren.

Ein erster, simpler Ansatz ist dabei neben dem Wert der Materialien auch die möglichen Angriffe auf Figuren mit in die Evaluation mit einzubeziehen. Auch dabei gilt es den Wert der angegriffenen Funktion mit einzuberechnen sowie die Anzahl der angreifenden Figuren. Am Ende müssen diese Werte sowohl für von schwarz angegriffene als auch von weiß angegriffene Figuren berechnet und voneinander subtrahiert werden, um den Vorteil im Angriff berechnen zu können. Diese Möglichkeit der Evaluation ist nur dann möglich, wenn es sich nicht um einen ruhigen Zustand handelt, wie in Kapitel 2.5 erklärt.

Ein weiterer, wichtiger Ansatzpunkt für aufbauende Angriffe ist die Kontrolle des Zentrums. beim Zentrum handelt es sich dabei um die Felder d4, d5, e4 und e5. Dies ist ein wichtiger Ansatz bei in Kapitel 2.6.1 beschriebenen "Opening-books", sowie bei in Kapitel 2.6.2 beschriebenen Matrizen zur Bewertung der Positionierung der Figuren auf einem Schachbrett. Um dies dennoch extra zu evaluieren, können alle Figuren gezählt werden, die das Zentrum erreichen können, mit deren Werten multipliziert werden und anschließend die Differenz der weißen sowie schwarzen Figuren berechnet. [?]

Um Angriffe effizient aufbauen zu können, ist auch Mobilität ein wichtiger Faktor im Spiel. In einer Studie von Eliot Slater wurde deutlich, dass die Anzahl der durchführbaren Züge bei gleichzeitiger Gleichheit der verfügbaren Materialien eine deutliche Korrelation mit der Anzahl der gewonnenen Spiele zeigt. [?]

Der simpelste Ansatz, um diese messbar zu machen, ist schlicht den benannten Wert der Anzahl der Züge zur Hilfe zu nehmen und diese für beide Spieler zu vergleichen.

Dies kann jedoch noch um verschiedene Faktoren erweitert werden. Dies ist beispielsweise Vorwärtsbewegungen höher zu bewerten als Rückwärtsbewegungen oder vertikale Bewegungen höher als horizontale. Ebenso können auch Züge mitgezählt werden, die auf ein Feld führen, das bereits von einer befreundeten Figur belegt ist. Dies stellt zwar keinen legalen Zug dar, sichert jedoch die befreundete Figur ab. [?]

Zudem kann zur Bewertung eines Angriffs auch die Sicherheit des gegnerischen Königs zur Hilfe genommen werden. Wie diese berechnet wird, ist in Kapitel 2.6.3 aufgeführt.

All diese Kriterien führen zu einer guten Möglichkeit Angriffe beziehungsweise den Aufbau von Angriffen bewerten zu können und gehen somit über die beschriebene Defensivstrategie hinaus.

## 2.6.5 Spielverlauf

Neben den bisher beschriebenen heuristischen Funktionen zur Bewertung eines Spiels, kann auch statistische Bewertung Einfluss auf diese nehmen.

Dazu ist es wichtig, eine möglichst große Historie an Spielen aufzubauen. Dazu kann im Laufe eines Spiels jeder einzelne Zustand abgespeichert werden, so dass dieser eindeutig identifizierbar ist. Am Ende des Spiels muss dazu dann ein Wert hinzugefügt werden, der Aufschluss über Sieg, Unentschieden oder Niederlag gibt. Dazu können die Werte 1, 0 und -1 dienen.

Beim Zusammenfügen der Historie des aktuellen Spiels mit der aufzubauenden Gesamthistorie aller Spiele gilt es dann über die Historien zu iterieren und nach gleichen Zuständen zu suchen. Diese gilt es dann zu migrieren, um eine Statistik aufzubauen, die Aufschluss über die Anzahl aller Siege und Niederlagen ausgehend von dem speziellen Zustand zu gewinnen.

Die Aussagekraft ist vor Allem dann kräftig, umso höher die Zahl ist. Wenn es sich um lediglich eine positive Bilanz von ein oder zwei Siegen mehr als Niederlagen handelt, kann dies schlicht an der geringen Zahl an Vorkommnissen des Zustands liegen. Ist dieser jedoch sehr hoch, ist ein Zufall eher unwahrscheinlich.

Deshalb gilt es eine möglichst große Historie aufzubauen, die vor Allem in der Anfangsphase, in der noch nicht so viele verschiedene Zustände erreichbar sind, guten Aufschluss über die Qualität des Zustands geben kann. Umso fortgeschrittener jedoch das Spiel ist, desto schwieriger wird es eine angemessene Anzahl an Iterationen über die speziellen Zustände zu erreichen, da diese immer häufiger vorkommen.

Da im Spiel Schach  $10^{40}$  verschiedene Spielzustände existieren [?] , ist es kaum möglich alle Zustände in angemessener Anzahl durchgegangen zu sein und abzuspeichern. Selbst unter der Annahme, dass ein Zustand inklusive dessen Bilanz in nur 20 Bit gespeichert werden kann (12 Bits für Schachspiel [?] und 8 Bits für Bilanz), würde dies  $10^{40} * 20 \text{ Bits} = 2 * 10^{41} \text{ Bits} = 2,5 * 10^{40} \text{ Bytes} = 2,5 * 10^{25} \text{ Petabytes}$  an Speicherplatz benötigen, was eine unverhältnismäßige Größe darstellt. Zudem würde die Ansammlung dieser Daten unverhältnismäßig lange dauern.

Aus diesem Grund stellt dies zwar eine gute Größe für den Beginn des Spiels dar, verliert jedoch im späteren Spielverlauf auf Grund fehlender Vorkommnisse der Zustände an Aussagekraft.

Computer Chess Compendium

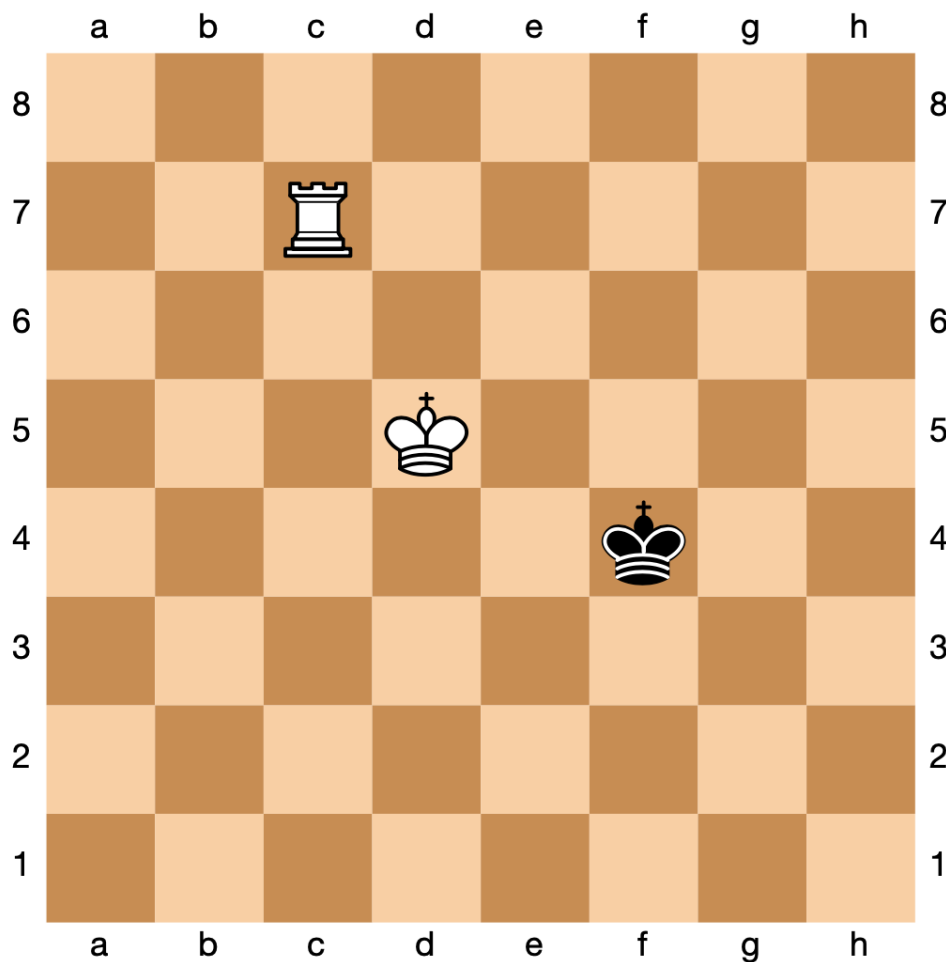


## 2.6.6 Finishing strategy

Nach dem Eröffnungsspiel und dem Mittelteil einer Schachpartie, folgt das Endspiel. Das Endspiel beschreibt dabei eine Situation, bei der sich nur eine geringe Anzahl an schwarzen und weißen Figuren auf dem Schachbrett befinden.

Zum zielstrebigem Gewinnen des Spiels, ohne überflüssige Züge, ist es hilfreich sich grundlegende Muster anschauen, die es ermöglichen den Gegner Schachmatt zu setzen. Neben der Möglichkeit zielstrebig und effizient das Spiel gewinnen zu können, kann das Wissen über solche Muster auch dazu beitragen den Gegner vom Gewinnen abzuhalten, da ersichtlicher wird was dessen Strategie ist.

In Abbildung 2.7 findet man eine simple Endspielsituation vor, in der die Partei mit den weißen Figuren die gegnerische Partei trivial Schachmatt setzen kann, da die schwarze Partei nur noch den König als Figur besitzt und sich einem Turm und einem König gegenüber sieht.



Endspielsituation RKvk

Ein Ausweg aus der drohenden Niederlage ist die 50-Züge-Regel, mittels der die schwarze Partei ein Remis erzielen kann. Für die Erfüllung dieser Regel muss nachgewiesen werden, dass in den letzten 50 Zügen keine Figur geschlagen oder ein Bauer gezogen wurde.[?] Um das Spiel zu gewinnen muss somit die weiße Partei zielstrebig den schwarzen König Schachmatt setzen.

Im Laufe der Jahre seitdem es das Schachspiel gibt wurden bereits einige Endspiele analysiert und evaluiert um daraus mögliche Muster und Strategien zu erhalten, die zu einem Sieg in einem Endspiel führen können. Diese vorliegenden Analysen helfen bei der Umsetzung einer Schach-KI, da mittels solcher Informationen schnell evaluiert werden kann welcher Zug am Gewinnbringendsten für den Spieler ist.

Theoretisch lässt sich auch durch die Verwendung von Suchalgorithmen, wie das bereits vorgestellte Alpha-Beta-Pruning, der beste Zug ermitteln, jedoch kann dies deutlich länger dauern als das Zurückgreifen auf bereits durchgeführte Analysen. Dies hat den Grund, da beispielsweise bereits bei drei verschiedenen Figuren, die sich zu Ende des Spiels auf dem Brett befinden, eine sehr große Anzahl an verschiedenen Zugmöglichkeiten ergibt. Dies ist unter anderem darin begründet, dass diese drei verbliebenen Figuren aus einer unterschiedlichen Konstellation aus Steinen bestehen. Neben den beiden Königen kann der dritte Stein jeder beliebige andere als ein König sein. Dabei hat diese dritte Figur pro Stein unterschiedliche Sprungeigenschaften, wie bspw. Distanz und Richtung. Ebenfalls hat die Position, auf der sich die Figuren befinden einen großen Einfluss auf die Möglichkeiten. Somit ergibt sich in Summe für ein Endspiel mit drei Figuren bereits eine Anzahl von 368.079 (siehe 2.6.6) verschiedenen Positionen.

Anzahl der Figuren	Mögliche Positionen
2	462
3	368.079
4	125.246.598
5	25.912.594.054
6	3.787.154.440.416
7	423.836.835.667.331
8	38.176.306.877.748.245 [?]

Zum aktuellen Zeitpunkt wurden alle möglichen Züge der letzten sieben Steine auf einem Schachbrett berechnet und liefert somit der zu programmierenden KI eine Datenbank mit vielen verschiedenen Zügen und den jeweils besten davon.

## 3 Technische Grundlagen

### 3.1 Kriterienformulierung

### 3.2 Python-Chess Evaluierung

In diesem Notebook wird die "python-chess-library", oder auch "chess core" genannt, Bibliothek evaluiert. Der "chess core" Funktionen zum Erstellen des Schachbretts, berechnen der erlaubten/möglichen Züge, durchführen der Züge etc. bereitstellen.

Dabei wird jede notwendige Funktion getestet und dessen Realisierung dokumentiert. Alle notwendigen Funktionen sind folgend gelistet.

- Schachbrett erstellen
- Schachbrett als ASCII ausgeben
- Züge auf dem Schachbrett ziehen
- Erlaubte Züge berechnen
- Auf Schach/Schachmatt prüfen
- Testen, ob Rochade, En Passant als erlaubte Züge gelistet werden
- *(Optional)* Schachbrett als SVG in JupyterNotebook ausgeben
- *(Optional)* Möglichkeiten evaluieren, Schachbrett je nach Positionierungen eindeutig identifizierbar in CSV zu schreiben

Des Weiteren werden folgende Aktionen/Berechnungen auf deren Umsetzbarkeit getestet. Dabei wird herausgefunden werden was nötig ist, um die aufgelisteten Aktionen mit der Bibliothek durchzuführen.

- Berechnen & Ausgeben erlaubter Züge für eine spezielle Figur
- Abwechselnder User & KI Input auf demselben Board
- Berechnen des Wertes des Schachbretts und der attackierten Figuren
- Speichern von Zügen/Entwicklung eines Schachbretts inklusive Sieg/Niederlagen als CSV-Datei

### 3.2.1 Voraussetzungen

Folgend werden die allgemeinen Voraussetzungen für die Verwendung und Installation der "python-chess-library" erläutert.

- Python 3 Da die zu verwendende Bibliothek auf Python 3 basiert muss diese Version auf dem auszuführenden Computer vorhanden sein.
  - [macOS](#)
  - [Linux](#)
  - [Win](#)
- Jupyter Notebooks Damit die aufgeführten Scripts direkt im Browser ausgeführt werden können haben sich die Autoren dieser Arbeit darauf verständigt Jupyter Notebook zu nutzen. Dabei bietet diese Applikation den Vorteil, dass Code und Dokumente live geteilt werden können und der entwickelte Code sofort ausgeführt werden kann. Ebenfalls bietet Jupyter Notebook die Möglichkeit unter Anderem Daten zu visualisieren.
  - [Installationsanleitung](#)
- "python-chess-library"
  - `pip install python-chess` Im weiteren Verlauf dieses Notebooks werden zusätzliche Module, wie beispielsweise Pandas, genutzt, die ebenfalls auf dem auszuführenden Rechner installiert sein müssen.

### 3.2.2 Erstellen eines Schachbretts und ausgeben als ASCII

Zu Beginn muss die Python-Chess-Library eingebunden werden, die für den weiteren Verlauf der Evaluierung und Implementierung benötigt wird.

```
In [3]: import chess
```

Das unten einzusehende Code Snippet zeigt, wie ein neues Schachbrett mit standardmäßigen Positionierungen der Figuren erstellt werden kann. Dies ist umzusetzen durch den Aufruf der `chess.Board()` Funktion und das Speichern der Rückgabe dieser Funktion in einer `board` Variable. Dieses `board` beinhaltet die Positionierungen aller Figuren und kann mittels der python-eigenen `print()` Funktion als ASCII Code ausgegeben werden, wie unten einzusehen ist. Zuletzt wird noch die Möglichkeit aufgezeigt, die auf einem spezifizierten Feld befindliche Figur auszulesen. Dieses Feld wird dabei über das Datenfeld `chess.B1` aufgerufen.

```
In [4]: board = chess.Board()

print ("\nBoard:")
print (board)

print ("\nPiece at B1:")
print (board.piece_at(chess.B1))
```

Board:

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

Piece at B1:

N

Dabei ist einzusehen, dass die Figuren durch einen Buchstaben abgekürzt werden. Figuren des ersten Spielers werden als Kleinbuchstaben dargestellt, Figuren des zweiten Spielers als äquivalente Großbuchstaben. Leere Felder werden mit einem Punkt dargestellt. Aus folgender Liste kann die Zuordnung der Buchstaben zu den Schachfiguren entnommen werden:

- p/P: Pawn / Bauer
- r/R: Rook / Turm
- n/N: Knight / Springer
- b/B: Bishop / Läufer
- q/Q: Queen / Dame
- k/K: King / König

Die Zuordnung der Felder findet mittels einer Kombination aus einem Buchstaben und einer Zahl statt. Die Buchstaben A-H geben dabei die horizontale Reihe an, die Zahlen 1-8 die vertikalen Reihen. Jedem Feld wird eine namensäquivalente Variable in der statischen Klasse "chess.B1" zugeordnet, wie beispielhaft in Zeile 16 im oberen Code Snippet erkannt werden kann.

### 3.2.3 Alternierende Eingaben von Nutzer & KI

Damit ein Schachspiel zustande kommen kann, ist es zwingend erforderlich, dass Nutzer und KI abwechselnd Züge auswählen und zum Board "pushen" können. Dafür muss zunächst überprüft werden, welcher Spieler an der Reihe ist und dieser muss einen Zug auswählen können, der daraufhin von dem Schachbrett übernommen wird. Auf diesem Schachbrett kann dann der nächste Spieler seinen Zug auswählen.

Bei der *"python-chess"* Bibliothek kann der zu agierende Spieler über die `board.turn` Variable ermittelt werden. Diese steht auf `True` wenn Spieler 1 an der Reihe ist und auf `False`, wenn Spieler 2 den nächsten Zug machen muss. Beim Ausführen einer `push` Operation auf dem `board`, wechselt die Variable automatisch ihren Wert.

Damit das Spiel erkennt, wann dieses vorbei ist, bietet die Bibliothek die `board.is_game_over()` Funktion. Diese gibt den Wert `True` zurück, falls das Spiel auf Grund eines Schachmatts oder anderer spielbeendender Umstände vorbei ist.

Um alle möglichen Züge ausgeben zu können, bietet die verwendete Bibliothek die Funktion `board.legal_moves`. Diese gibt alle gültigen Züge nach *chess960* Standard aus. Dabei werden nur die Züge ausgegeben, die zum einen vom aktuellen Schachbrett aus durchführbar sind und zum anderen nicht zu einer unmittelbaren Niederlage führen. Das bedeutet, dass bei den legalen Zügen keine Züge ausgegeben werden, bei der sich der Spieler beispielsweise selbst ins Schach stellt oder ein durch den Gegner verursachtes Schach ignoriert. Mittels der `board.uci()` Funktion können diese in die, aus 4 Zeichen bestehende, leserlichere Form gebracht werden. Die ersten beiden Zeichen stellen dabei das startende Feld dar, während die zweiten zwei Zeichen das Feld darstellen, auf das sich die Figur vom Startfeld bewegt.

Diese Funktion kann beim Zug des Nutzers verwendet werden, um diesem eine einfachere Übersicht über seine Möglichkeiten zu geben und den Zug in einer ihm verständlichen Form einzulesen.

Mittels der `list()` Funktion können die legalen Züge zu einer Liste zusammengefasst werden.

Im folgenden Beispiel sind alle notwendigen Schritte für ein Schachspiel zwischen Nutzer und KI erkennbar. Die KI ermittelt dabei ihren Zug durch eine zufällige Auswahl aus der Liste aller legalen Züge. Dabei wurden die einzelnen Methoden, wie oben beschrieben, implementiert und genutzt.

Zusätzlich müssen einigen Funktionen der Python-Chess-Library importiert werden, die für die Visualisierung eines Schachbretts als SVG benötigt werden. Ebenfalls wird das Modul `random` eingebunden, da es für die Berechnung zufälliger Züge benötigt wird.

```
In [5]: import chess.svg
import random
board = chess.Board()
```

`get_random_move` wählt zufällig einen Zug aus den erlaubten Zügen aus und gibt diese als `CHESS.Move` zurück. Hingegen wird die Funktion `get_legal_moves_uci` genutzt um die erlaubten Züge berechnen zu lassen.

```
In [6]: def get_random_move():
return random.choice(list(board.legal_moves))

def get_legal_moves_uci():
return list(map(board.uci, board.legal_moves))
```

Die Funktion `get_user_move` wird verwendet um den Nutzer die möglichen, bzw. erlaubten, Züge auszugeben. Dabei wird die zuvor eingeführte Funktion zur Berechnung von legalen Zügen verwendet.

```
In [7]: def get_user_move():
print("Possible Moves: ")
print(get_legal_moves_uci())

print("Enter your move:")
move = input()

return chess.Move.from_uci(move)
```

Der folgende Codeausschnitt wird genutzt um einen Nutzer gegen den Computer spielen zu lassen, der jeweils einen zufälligen Zug verwendet. Hierbei wird für das Jupyter Notebook ein Counter eingeführt, damit das Spiel bereits nach zwei Zügen pro Spieler beendet ist. Alternativ dazu wird das Spiel beendet, sobald ein Sieger oder ein Patt feststeht.

Ebenfalls wird als Unterstützung für den Spieler pro Zug das Schachbrett und die legalen Züge ausgegeben.

```
In [8]: counter = 0
while (not board.is_game_over() and counter < 4):
print("-----")
```



```
print(board)
print("-----")
print()

if board.turn:
    board.push(get_user_move())
    print("Your Move: ")
else:
    board.push(get_random_move())
    print("AIs Move:")

counter+=1
```

```
print(board)
print("[...]")
```

```
-----
r n b q k b n r
P P P P P P P P
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
-----
```

Possible Moves:

['g1h3', 'g1f3', 'b1c3', 'b1a3', 'h2h3', 'g2g3', 'f2f3', 'e2e3', 'd2d3', 'c2c3', 'b2b3']

Enter your move:

c2c3

Your Move:

```
-----
r n b q k b n r
P P P P P P P P
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . P . . . . .
```

```
P P . P P P P P
R N B Q K B N R
```

AI's Move:

```
-----
r . b q k b n r
p p p p p p p p
. . n . . . . .
. . . . . . . .
. . . . . . . .
. . P . . . . .
P P . P P P P P
R N B Q K B N R
-----
```

Possible Moves:

['g1h3', 'g1f3', 'd1a4', 'd1b3', 'd1c2', 'b1a3', 'c3c4', 'h2h3', 'g2g3', 'f2f3', 'e2e3']

Enter your move:

d1a4

Your Move:

```
-----
r . b q k b n r
p p p p p p p p
. . n . . . . .
. . . . . . . .
Q . . . . . . .
. . P . . . . .
P P . P P P P P
R N B . K B N R
-----
```

AI's Move:

```
r . b q k b n r
. p p p p p p p
p . n . . . . .
. . . . . . . .
Q . . . . . . .
. . P . . . . .
```

```
P P . P P P P P
R N B . K B N R
[...]
```

### 3.2.4 Ausgeben des Boards als SVG

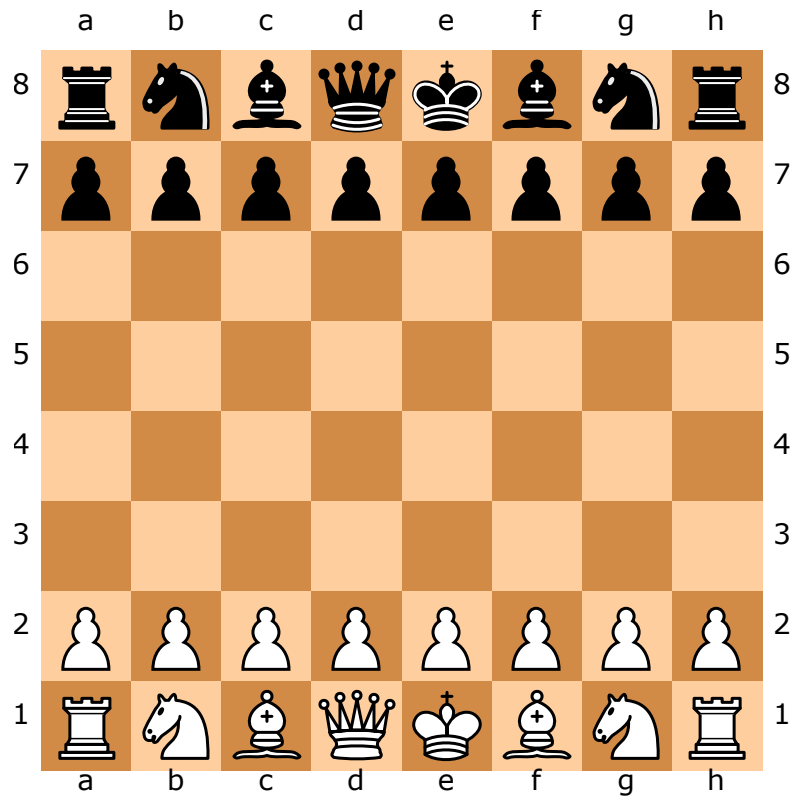
Um eine bessere Darstellung und Erklärung der Implementation in der theoretischen Ausarbeitung zu ermöglichen, wäre eine visuelle Darstellung des Schachbretts wünschenswert. Die "python-chess-library" ermöglicht dies durch ein Konvertieren des Boards zu einer SVG-Datei. Diese kann dann mittels der python-eigenen `SVG()` Funktion im Python-Notebook angezeigt werden.

Um dies zu ermöglichen, muss zunächst die SVG library aus *IPython* importiert werden. Anschließend kann das Board über die `chess.svg.board()` Funktion konvertiert und anschließend als SVG ausgegeben werden.

```
In [9]: from IPython.display import SVG

        board = chess.Board()
        SVG(chess.svg.board(board=board))
```

Out [9]:

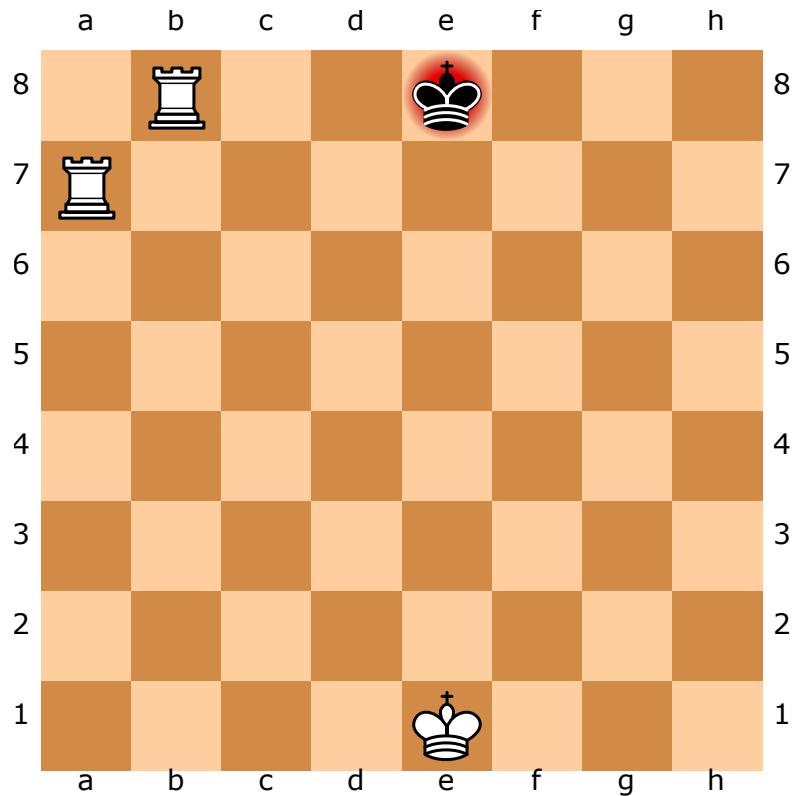


Die verwendete Bibliothek bringt den Vorteil mit sich, dass einige wichtige Eigenschaften während eines Spiels visualisiert werden können. Ein Beispiel für diese Visualisierung ist das Schachmatt, bei dem der betreffende König rot hervorgehoben wird. Ein weiteres Beispiel zum Thema Visualisierung wird im folgenden Kapitel gegeben.

Um diesen Fall simulieren zu können, wird in diesem Beispiel eine bestimmte Situation während eines Schachspiels importiert, in der ein Schachmatt vorliegt. Das Schachmatt auf der Position E8 wird in diesem Codeausschnitt manuell, durch den Übergabeparamter `chess.E8`, hervorgehoben.

```
In [10]: board = chess.Board("1R2k3/R7/8/8/8/8/8/4K3 b KQkq - 0 1")
          SVG(chess.svg.board(board, check=chess.E8))
```

Out [10]:



### 3.2.5 Berechnen & Ausgeben legaler Züge eines speziellen Felds

Für eine bessere, visuelle Darstellung in der theoretischen Ausarbeitung wurde die Möglichkeit geprüft spezielle Felder auf dem ausgegebenen SVG markieren zu können. Solche Felder können beispielsweise die errechneten erlaubten Züge einer speziellen Figur / eines speziellen Felds darstellen, um anzuzeigen, welche Möglichkeiten eine Figur in der aktuellen Situation besitzt.

Dies kann realisiert werden, indem auf die legalen Züge zurückgegriffen wird und diese gefiltert werden. Bei dem verwendeten Filter müssen die Ausgangspositionen, die mit `move.from_square` ausgelesen werden können, mit dem eingegebenen Feld übereinstimmen.

```
In [11]: board = chess.Board()

print("Please enter field:")
field = input()
```

```
moves_from_spec_field = list(filter(lambda move: move.from_square is chess.S
```

Please enter field:

a2

Diese herausgefilterten Züge werden dann zu deren Zielfelder zugeordnet. Diese können mit der Funktion `move.to_square` ausgelsen werden.

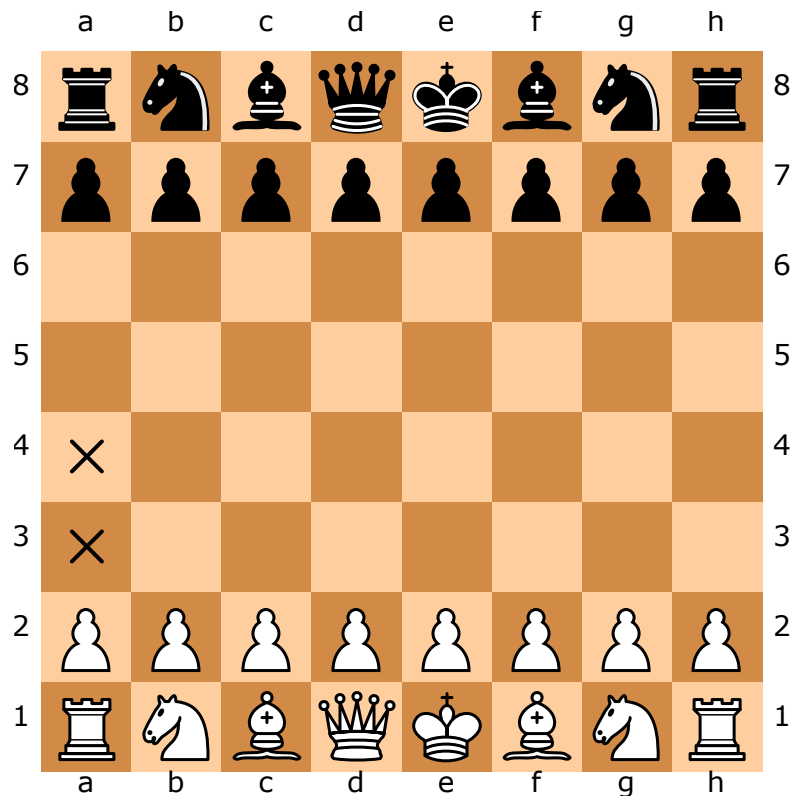
```
In [12]: square_nums = list(map(lambda move: move.to_square, moves_from_spec_field))
```

Nun kann ein `SquareSet` erstellt und alle gefilterten Felder diesem hinzugefügt werden. Das `SquareSet` kann dann beim Erstellen des Board angegeben werden. Dies veranlasst das Zeichnen von Kreuzen auf den berechneten Feldern, die von dem angegebenen Feld aus erreicht werden können.

```
In [13]: squares = chess.SquareSet()
         for square_num in square_nums : squares.add(square_num)

SVG(chess.svg.board(board=board, squares=squares))
```

Out [13]:



### 3.2.6 Speicherung der einzelnen Spielzüge in history.csv

Um die KI Entscheidungen um ihre Spielzüge auch von vorherigen Spielen und desse Ausgängen abhängig zu machen, muss eine Historie angelegt werden, die die Spielbretter und den Ausgang des Spiels im Nachgang beinhaltet. Anhand dieser Historie kann die KI in zukünftigen Spielen dann den Wert des Spielbrettes abschätzen und das Spielbrett evaluieren, das die höchste Chance auf einen Sieg bietet.

Zur Erweiterung der Liste wird zuerst das Modul Pandas genutzt, das es ermöglicht Informationen als Datenbank in einer Variable zu speichern. Ebenso wird angegeben unter welchem Pfad sich die Zughistorie befinden soll.

```
In [14]: import pandas as pd
        HISTORY_FILE_LOC = "res/history.csv"
```

Dazu kann testweise ein Spiel durchgespielt werden, wobei jeder Zug zufällig aus der Liste der legalen Züge gewählt wird.

```
In [15]: def get_random_move(board):
        return random.choice(list(board.legal_moves))
```

Nach jedem Zug wird das Spielbrett in der "FEN" Darstellung einer Liste hinzugefügt. Die "FEN" Darstellung ist eine gekürzte Form der Darstellung des Spielbretts, wobei das Spielbrett dennoch eindeutig identifizierbar bleibt. Dabei wird neben dem aktuellen Spielbrett auch die Farbe der nächste zu spielenden Figur, die Anzahl der Züge beider Seiten und weitere Informationen angegeben. Um diese aus der Darstellung zu kürzen, wird die auf die "FEN" Darstellung des Boards die `split` Funktion angewandt und nur das erste Element aus der daraus entstehenden Liste gespeichert, da die einzelnen Merkmale in dieser Darstellung per Leerzeichen getrennt werden.

```
In [16]: def play_random_chess_game(board):
        turn_list = list()
        while not board.is_game_over():
            turn_list.append(board.fen().split(" ")[0])
            board.push(get_random_move(board))
        return turn_list
```

Nach Ende des Spiels wird über der zu spielenden Farbe ermittelt, wer der Sieger des Spiels ist und dementsprechend der Wert "+1" oder "-1" zurückgegeben. Mit diesem wird dann eine Key-Value Liste erstellt, die zu jedem Spielbrett des vergangenen Spiels den Wert zuweist, der Aussage über den Sieger gibt.

Diese Liste wird dann zusammen gefügt mit der bisher vorhandenen Historie. Falls bereits ein Eintrag für ein gleichartiges Schachbrett existiert, wird der Siegwert zu dem aktuellen Wert aus der Historie dazu addiert, andernfalls wird ein neuer Eintrag mit dem Siegwert angelegt.

```
In [17]: def store_game(turn_list, victory_status):
            new_turn_dict = dict.fromkeys(turn_list, victory_status)
            # get existing board history
            history = pd.read_csv(HISTORY_FILE_LOC)
            history_dict = dict(zip(list(history.board), list(history.value)))
            # merge existing history with new boards and sum the victory states
            merged_history_dict = { k: new_turn_dict.get(k, 0) + history_dict.get(k, 0) }
            merged_history = pd.DataFrame(list(merged_history_dict.items()), columns=['board', 'value'])
            # overwrite history csv with new, merged history
            merged_history.to_csv(HISTORY_FILE_LOC)
```

Der Siegwert nach dem Schachbrett gibt dann, je nach Höhe, Auskunft darüber, wie wahrscheinlich es ist mit einem solchen Schachbrett zu gewinnen. Umso höher der Wert in den positiven Bereich fällt, desto wahrscheinlicher ist ein Sieg für Spieler 1. Umso höher der Wert in den negative Bereich fällt, desto besser sind die Aussichten für Spieler 2.

Diese Erkenntnis kann dann von der KI genutzt werden, um beim "iterative Deepening"-Prozess alle möglichen Schachbretter zu evaluieren und sich so für den besten Zug zu entscheiden.

Der folgende Code koordiniert die Abläufe der Funktionen um die Datei `history.csv` zu erweitern.

```
In [ ]: for i in range(0, 10):
            i += 1
            board = chess.Board()
            turn_list = play_random_chess_game(board)

            victory_status = 1 if board.turn == 'w' else -1
            store_game(turn_list, victory_status)
            print("Games finished")
```

Ein Ausschnitt aus der `history.csv` ist im folgenden Snippet zu sehen.



```
In [ ]: df = pd.read_csv(HISTORY_FILE_LOC)
        df.head()
```

Damit eingeschätzt werden kann, wie aussichtsreich ein bestimmter Boardzustand ist, kann ein gespeicherter Spielverlauf zur Hilfe genommen werden. Mit Hilfe der Funktion `compare_board_history` kann ein Erwartungswert berechnet werden. Umso höher der zurückgegebene Wert der Funktion, desto wahrscheinlicher ist ein Sieg für Weiß. Hingegen ist ein Sieg für Schwarz umso wahrscheinlicher, sobald der Wert deutlicher in den negative Bereich fällt.

Der Vorgang zur Berechnung dieses Wertes beginnt mit dem Einlesen der `history.csv` in ein Pandas Dataframe. Daraufhin wird durch die Reihe des Dataframes iteriert, die die Schachbretter in FEN-Schreibweise enthält. Sobald die aktuelle Spielsituation in der Historie gefunden wurde, wird deren Wert ausgelesen und zurückgegeben. Falls die aktuelle Spielsituation nicht vorhanden ist, wird der Wert 0 zurückgegeben.

```
In [ ]: def compare_board_history(board):
        df = pd.read_csv(HISTORY_FILE_LOC)
        row = df.loc[df['board'] == board.fen().split(" ")[0]]
        value = row['value'].item() if len(row['value']) == 1 else 0
        return value
```

### 3.2.7 Überprüfung auf ein Schach oder Spielende

Damit der Spielverlauf korrekt nach den Regeln des Spiels verlaufen kann, muss während jedes Zuges darauf geprüft werden, ob ein Schach, Schachmatt oder Patt vorliegt. Schach ist hierbei eine Stellung während des Spiels, bei dem der König in Bedrängnis geraten ist. Ein Schach kann im weiteren Verlauf zu einem Schachmatt führen. Hierbei liegt der Unterschied darin, dass der Spieler, der sich im Schachmatt befindet mit keinem regelkonformen Zug sich aus dem Schachmatt befreien kann und somit das Spiel verloren hat. Steht der König Schach kann er mit einem gezielten Zug sich aus dieser Lage befreien. Ein Patt ist eine Endposition beim Schach, die die Eigenschaft hat, dass keiner der beiden Spieler ein Schachmatt erreichen kann.

Da ein Schachmatt oder Patt das Schachspiel beendet ist diese Prüfung ein essenzieller Aspekt der zu entwickelnden KI. Um diese Prüfung durchzuführen bietet die verwendete Bibliothek bereits einige Funktionen. Ebenso ist es wichtig zu überprüfen, ob ein Schach vorliegt, da der Spieler zuerst dies lösen muss, bevor er weiterspielen kann. Um diese Prüfung durchzuführen steht jeweils für die Überprüfung eines Schachs oder Spielendes eine Funktion zur Verfügung, die das aktuelle Spiel auf diesen Aspekt überprüft. Um

herauszufinden, ob es sich um ein Schach handelt, kann die folgende Funktion verwendet werden. \* `is_check()` \* Überprüft die aktuellen Begebenheiten auf ein mögliches Schach

Hingegen gibt es die Funktion `is_game_over`, die die gespielten Züge auf jegliche Arten einer benötigten Beendigung des Spiels überprüft. Hierbei beinhaltet `is_game_over` einige mögliche Überprüfungen auf Schachmatt oder Patt. \* `is_game_over()` \* Stellt sicher, ob das Spiel auf Grund eines Schachmatts oder anderer spielbeendender Umstände vorbei ist \* Hierbei wird überprüft, ob ein Schachmatt (`is_checkmate()`), ein Patt (`is_stalemate()`), eine Tote Stellung (`is_insufficient_material()`), die 75-Züge-Regel (`is_seventyfive_moves()`), eine Figur fünfmal auf der gleichen Position sich befindet (`fivefold_repetition`), oder eine spezielle Endbedingung vorliegt.

Um herauszufinden, ob ein Spieler tatsächlich gewonnen hat kann eine Teilfunktion der zuvor genannten genutzt werden. Diese Funktion überprüft lediglich, ob ein Schachmatt vorliegt. \* `is_checkmate()` \* Stellt fest, ob es sich nach den bereits gespielten Zügen ein Schachmatt vorliegt

```
In [ ]: # import situation where checkmate is True
```

```
board = chess.Board("r1bqkb1r/pppp1Qpp/2n2n2/4p3/2B1P3/8/PPPP1PPP/RNB1K1NR b 1")
```

```
print("Check: ", board.is_check())
print("Checkmate: ", board.is_checkmate())
print("Game is over: ", board.is_game_over())
```

Im folgenden wird ein Beispiel einer Spielsituation gezeigt, in der ein Schachmatt vorliegt und die auf Schach, Schachmatt und Spielende überprüft wurde.

```
In [ ]: SVG(chess.svg.board(board=board))
```

### 3.2.8 Überprüfung, ob en passant und "Rochaden unterstützt werden

Im Schach gibt es einige spezielle Züge, die es ermöglichen eine Figur eine Aktion durchzuführen zu lassen, die normalerweise laut der grundlegenden Definition dieser nicht möglich ist.

Die Schachfigur eines Bauers darf normalerweise nur dann eine andere Figur schlagen, wenn sich diese direkt in dem diagonal vor dem Bauern angrenzenden Feld befindet. Durch die sogenannte Regel "en passant", im deutschen "im Vorbeigehen", wird diese Regel erweitert. Eingesetzt werden kann "en passant", wenn auf einen Bauern die Sonderregel des Doppelschritts aus der Grundstellung angewendet wird. Steht in diesem Fall der mit Doppelschritt herausgerückte Bauern neben einem des Gegners, dann kann dieser den neu

herausgerückten Bauern durch en passant schlagen. Hierbei springt der angreifende Bauer des Gegners direkt hinter den herausgerückten und schlägt ihn somit.

Damit die Schach-KI alle möglichen Züge des Gegners bedenken und ebenso alle Züge ausführen soll, muss überprüft werden, ob die verwendete Library die beiden Sonderzüge unterstützt, oder ob diese Unterstützung manuell entwickelt werden muss.

```
In [ ]: # import situation where en passant is possible
        board = chess.Board("rnbqkbnr/1pp1pppp/p7/3pP3/8/8/PPPP1PPP/RNBQKBNR w KQkq d5")

        # the function board.has_legal_en_passant() could check if a en passant is possible
        print("Library is supporting en passant: ", chess.Move.from_uci("e5d6") in board.legal_moves)
```

Die Chess-Core Bibliothek unterstützt somit den Sonderzug en passant.

Im Folgenden wird ein Schachbrett angezeigt, bei dem es dem weißen Bauer (e5) möglich ist im Vorübergehen den schwarzen Bauer (d5) zu schlagen.

```
In [ ]: squares = chess.SquareSet([chess.D6])
        chess.svg.board(board=board, squares=squares)
```

Neben dem en passant gibt es einen weiteren bekannten Sonderzug, die sogenannte Rochade. Bei der Rochade lassen sich die Positionen eines Turms und des Königs tauschen, wobei für diesen Vorgang nur ein Zug benötigt wird. Dabei ist zu beachten, dass die Voraussetzung für diesen Zug ist, dass sowohl der zu verwendende Turm, als auch der König im Verlauf des Spiels nicht genutzt wurden. Ebenfalls dürfen die Felder zwischen König und Turm nicht belegt sein und keines der Felder, durch die der König ziehen muss, darf durch eine gegnerische Figur bedroht sein, sowie der König vor und nach der Rochade nicht im Schach stehen.

Für jeden Spieler gibt es zwei verschiedene Möglichkeiten der Rochade, einerseits die kurze, als auch die lange Rochade. Ein Beispiel für eine lange Rochade der weißen Figuren ist, dass der Turm (a1) und der König (e1) ihre Positionen tauschen und somit der Turm sich auf dem Feld d1 und der König auf c1 befindet.

```
In [ ]: # import situation where castling is possible
        board = chess.Board("8/8/8/8/8/8/8/R3K3 w KQkq - 0 1")
        # shortcut for the castling-move with the queenside rook
        castling = "0-0-0"

        # check if castling is in the legal moves and print the result
        if castling in str(board.legal_moves):
```

```

print("Library is supporting castling: True")
board.push_san(castling)
else:
    print("Library is supporting castling: False")

SVG(chess.svg.board(board=board))

```

Eine durchgeführte kurze weiße Rochade, bei der **a1** und **e1** die Positionen getauscht haben, sodass der König sich nun auf **c1** und der Turm auf **d1** befindet. Die Abkürzung für eine kurze Rochade ist *O-O*, das für den Turm auf der Seite des Königs steht und *O-O-O* für eine lange Rochade, wobei die Abkürzung für den Turm auf der Seite der Dame steht.

### 3.2.9 Einbinden eines Opening-Books in die Chess AI

Polyglot ist ein Open-Source Format, in dem sogenannte Opening-Books erstellt werden können. Opening-Books sind Ansammlungen von Spielzügen, die im Schach zur Eröffnung genutzt werden können. Hierbei wird zum Einstieg des Spiels nicht auf eine Künstliche Intelligenz zurückgegriffen, sondern auf ein Verzeichnis von bereits bestehenden Zügen, die sich innerhalb dieses Opening-Books befinden. Der Vorteil eines Opening-Books liegt darin, dass bereits qualitativ hochwertige Strategien verfügbar sind und somit ein anspruchsvolles Spiel ermöglichen.

Sobald das Opening-Book, im Verlauf des Spiels, keinen passenden Zug als Antwort bereitstellen kann, übernimmt die Künstliche Intelligenz.

Für das Einbinden eines Opening-Books werden einige Funktionen aus der Core Chess Library genutzt und müssen somit eingebunden werden.

```
In [ ]: import chess.polyglot
```

Als nächstes wird ein neues Spiel/Board erzeugt, indem die Figuren standardmäßig angeordnet werden. Ebenfalls wird ein Opening-Book in eine Variable geladen, damit aus dieser der bestmögliche Zug ausgewählt werden kann. Dies geschieht, indem das Opening-Book das aktuelle Board übergeben bekommt und anhand der Positionen der Figuren einen passenden Zug auswählt.

Der ausgewählte Zug wird als nächstes auf dem Schachbrett ausgeführt und somit wird in diesem Beispiel der weiße Bauer von **e2** nach **e4** gezogen.

```
In [ ]: board = chess.Board()
        book = chess.polyglot.open_reader("res/polyglot/Performance.bin")

def get_move(board):
    # find the move with the highest weight for the current board
    try:
        main_entry = book.find(board)
        all_entries = book.find_all(board)

        move = main_entry.move()
        print("Selected move with the highest weight: ", move)
        print("All available moves for this situation: ", ", ".join([str(entry)
                                for entry in all_entries]))

        return move
    except IndexError:
        print("The opening book cannot find an appropriate move!")

for i in range (1,3):
    next_move = get_move(board)
    board.push(next_move)
    print("Selected move: {}".format(next_move))
book.close()

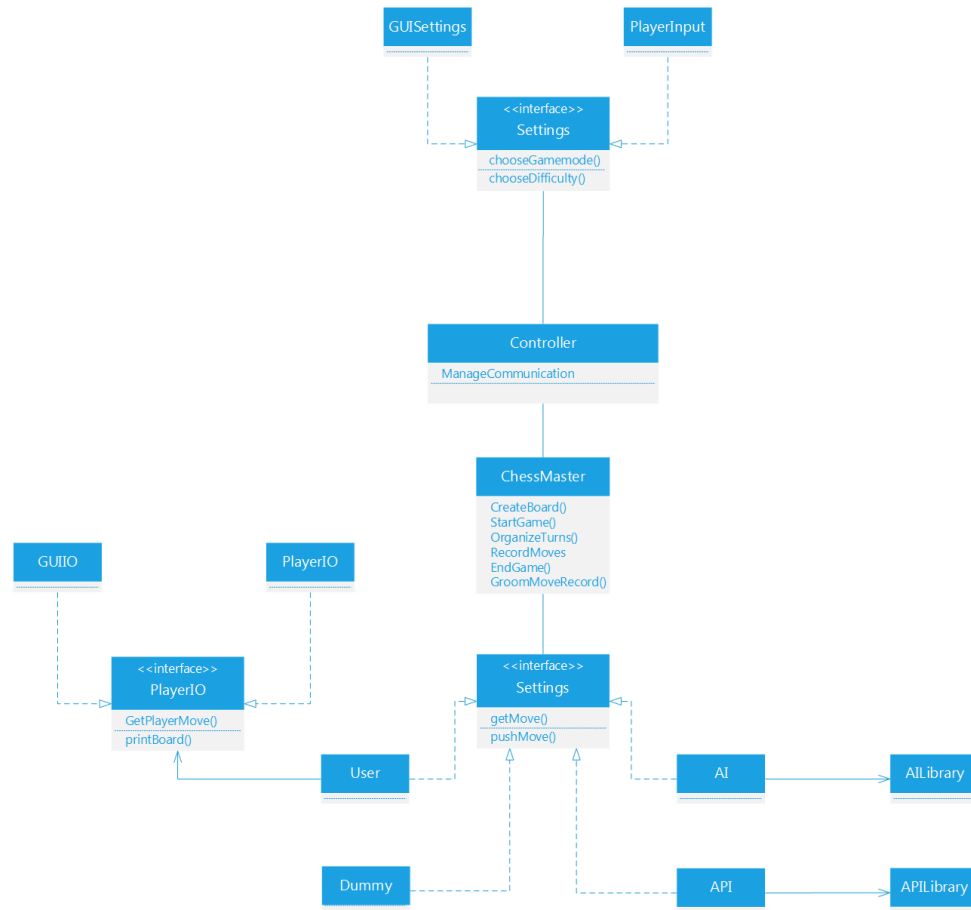
SVG(chess.svg.board(board=board))
```

### 3.3 Architektur

Um die Ziele der Erweiterbarkeit, Wartbarkeit und einfachen Verständlichkeit zu erfüllen, ist eine modulare Aufbauweise des Projektes von Nöten. Die auf diesen Prinzipien beruhende Architektur wird im folgenden Kapitel beschrieben.

Zunächst wurde das Programm der Aufgaben entsprechend in einzelne Klassen unterteilt, die verschiedene Aufgaben zugewiesen bekommen haben. Diese Aufteilung kann Figur 3.2.1 entnommen werden.

Als zentrales, Verbindungselement steht zunächst der “Controlle”. Dieser übernimmt die Abfrage aller Optionen, falls diese nicht beim Start des Programms mit übermittelt werden. Zudem initialisiert der Controller die Spielteilnehmer sowie den “ChessMaster”. Bei



Figur 3.2.1 Klassendiagramm Architektur ChessAI

letzterem übergibt der Controller diesem die initialisierten Spielteilnehmer. Die Aufgaben des “ChessMaster” wird später genauer beschrieben.

Zur Abfrage der Optionen steht dem Controller eine ‘Settings’ Schnittstelle zur Verfügung. Diese wird für Eingaben des Nutzers einerseits über eine Konsole als auch optional über eine grafische Benutzeroberfläche implementiert. Dabei wird abgefragt, welchen Typ die jeweiligen Spielteilnehmer annehmen sollen und es können zudem zusätzliche Optionen für die einzelnen Spieler definiert werden. Als Beispiel kann für Spieler 1 der Typ “User” gewählt werden und für Spieler 2 der Typ “AI”. Dies ermöglicht ein Spiel des Nutzers gegen die im Rahmen dieser Arbeit entwickelte KI. Für die KI wird darauf folgend noch der Schwierigkeitsgrad abgefragt. Zusätzlich kann für jeden Spieler ein Name festgelegt werden.

Die Aufgaben des “ChessMaster” erstrecken sich über die Verwaltung des Schachspiels an sich, das Ansprechen der jeweiligen Spieler zum Ermitteln ihrer Züge sowie dem Durchführen des gewählten Spielzugs auf dem Schachbrett. Zusätzlich speichert es jedes Schachbrett, das sich im Laufe des Spiels ergibt, und fügt diese zur “board\_history.” Datei hinzu. Diese speichert alle Schachbretter gemeinsam mit einem numerischen Wert. Dieser gibt Aufschluss über Erfolgsaussichten der jeweiligen Akteure des Spiels. Dazu wird nach

jedem Spiel zu dem entsprechenden Eintrag in der Datei eine eins zu dem alten Wert aufaddiert, wenn der Spieler der weißen Figuren das Spiel gewonnen hat und eine eins subtrahiert, wenn der Spieler der schwarzen Figuren das Spiel gewonnen hat. Bei einem Unentschieden bleibt der alte Wert bestehen. Dies hilft der KI bei der Bewertung eines Schachbretts unter zur Hilfenahme von statistischen Werten.

Dieser spricht die vom Controller erstellten Spieler an. Diese können, wie bereits angedeutet, von verschiedenen Typen sein. Zur Auswahl stehen

- User - Ein menschlicher Akteur kann Züge über eine Nutzerschnittstelle eingeben
- AI - Die künstliche Intelligenz versucht den best möglichen Zug zu berechnen
- Dummy - Ein zufälliger Zug wird ausgewählt
- (Optional) API - Ein Zug wird über eine Schnittstelle zu einer Online Schachplattform, auf der menschliche sowie künstliche Spieler teilnehmen dürfen, bestimmt

Dazu existiert eine “Player” Schnittstelle, die für jeden dieser Spieler implementiert werden muss.

Die “User” Implementation greift dabei zur Ermittlung des Zuges auf eine “PlayerIO” Schnittstelle zurück. Diese wiederum kann - ähnlich zur “SettingsUI” Schnittstelle - sowohl für Eingaben über eine Konsole als auch für Eingaben über eine grafische Nutzeroberfläche implementiert werden.

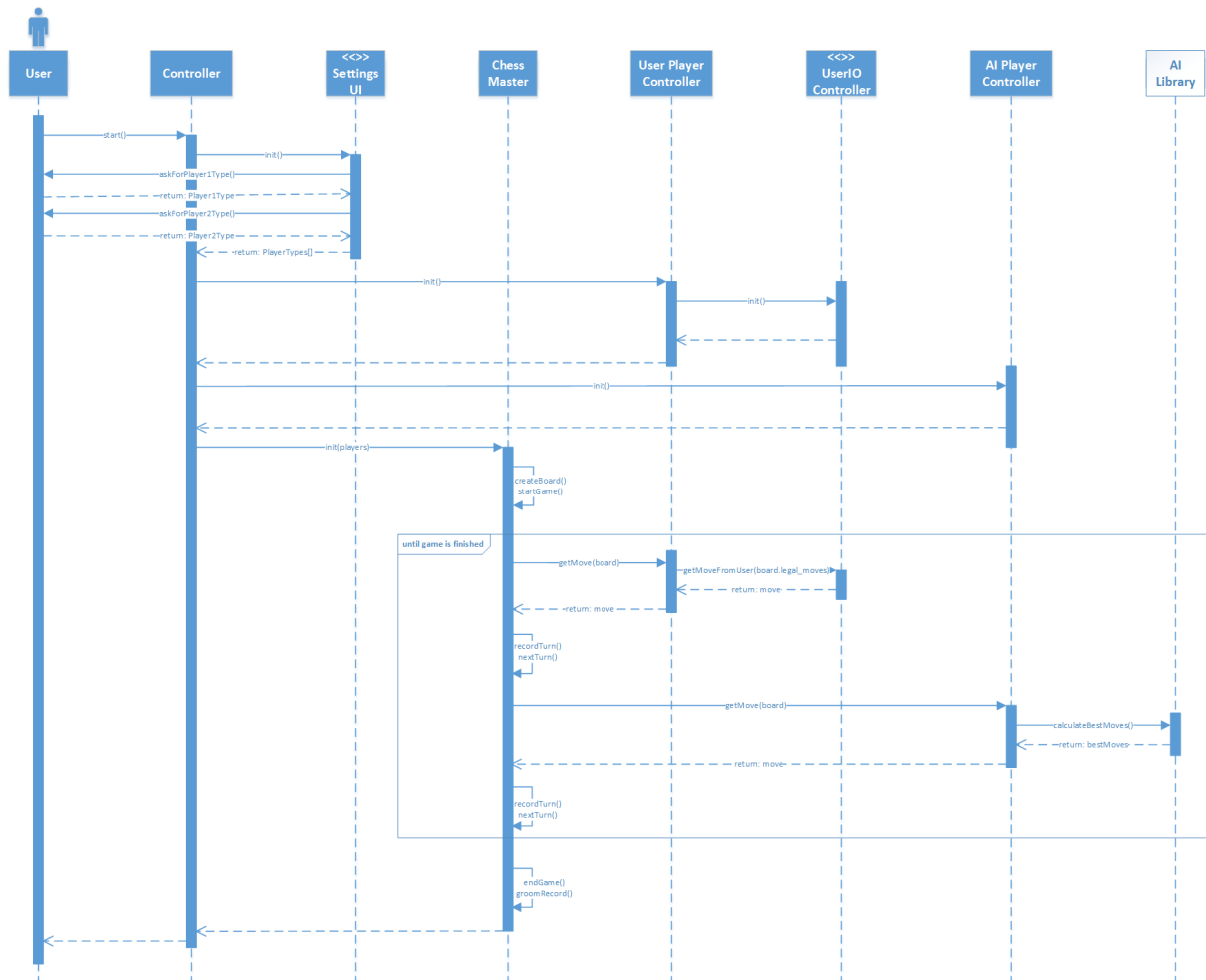
Die “AI” sowie die “API” Implementation greifen für die Ermittlungen ihrer Züge nochmals auf eigens erstellte Bibliotheken zurück, die elementare Funktionen erhalten.

Der genaue Ablauf der Ermittlung der Züge sowie der andere Operationen des Programmes werden in Kapitel XY.ZY näher erläutert.

In Figur 3.2.2 ist der sequentielle Ablauf der Funktionsaufrufe erkennbar.

Dabei wird zunächst über die “Settings” Schnittstelle nach den Spielertypen sowie Namen gefragt. initialisiert der “Controller” die Spieler, in diesem Fall einerseits den “User”, der wiederum die “UserIO” Schnittstelle implementiert, und außerdem den “AI” Player.

Darauffolgend spricht der “Controller” den “ChessMaster” an. Dieser startet nun das Spiel und fragt solange wie das Spiel nicht vorbei ist immer abwechselnd erst zu Spieler 1 - dem Nutzer - und dann zu Spieler 2 - der KI - nach dem nächsten Zug. Dabei übergibt der “ChessMaster” stets das aktuelle Schachbrett. Der Nutzer ermittelt den durchzuführenden Zug über eine Abfrage an den Nutzer über die entsprechende Schnittstelle, der “AI” Spieler, indem dieser Funktionen aus der entsprechenden Bibliothek zur Hilfe nimmt.



Figur 3.2.2 Sequenzdiagramm Architektur ChessAI

Nach jedem Zug fügt der “ChessMaster” diesen zum Schachbrett hinzu und speichert dieses in einen Spielverlauf. Nach Ende des Spiels wird dieser in das “board\_history.csv” eingepflegt wie weiter oben bei der Beschreibung der Klasse bereits erläutert.

Abschließend beendet der “ChessMaster” das Spiel, wenn keine Revance gewünscht ist, und so wird auch das Programm im Anschluss daran geschlossen.



## 4 Implementation

# **5 Evaluation**

## **5.1 Kriterienerfüllung**

## **5.2 Einordnung Intelligenz**

# Abkürzungsverzeichnis

**KI** Künstliche Intelligenz

# Appendices

---

**Algorithm 2** Minimax Algorithmus [?]

---

```
function minimax_decision(state)
  for all ainactions(state) do
    value  $\leftarrow$  min_value(result_state(state, a))
    if value > best_value then
      best_value  $\leftarrow$  value
      best_move  $\leftarrow$  a
    end if
  end for
  return best_move
end function

function max_value(state)
  if terminal_test(state) then
    return utility(state)
  end if
  v  $\leftarrow$   $-\infty$ 
  for all ainactions(state) do
    v  $\leftarrow$  max(v, min_value(result_state(state, a)))
  end for
  return v
end function

function min_value(state)
  if terminal_test(state) then
    return utility(state)
  end if
  v  $\leftarrow$   $\infty$ 
  for all ainactions(state) do
    v  $\leftarrow$  min(v, max_value(result_state(state, a)))
  end for
  return v
end function
```

---

---

**Algorithm 3** Alpha Beta Algorithmus [?]
 

---

```

function alpha_beta_search(state)
  for all ainactions(state) do
    value  $\leftarrow$  min_value(result_state(state, a),  $-\infty$ ,  $+\infty$ )
    if value > best_value then
      best_value  $\leftarrow$  value
      best_move  $\leftarrow$  a
    end if
  end for
  return best_move
end function

```

```

function max_value(state,  $\alpha$ ,  $\beta$ )
  if terminal_test(state) then
    return utility(state)
  end if
  v  $\leftarrow$   $-\infty$ 
  for all ainactions(state) do
    v  $\leftarrow$  max(v, min_value(result_state(state, a),  $\alpha$ ,  $\beta$ ))
    if v  $\geq$   $\beta$  then
      return v
    end if
     $\alpha$   $\leftarrow$  max( $\alpha$ , v)
  end for
  return v
end function

```

```

function min_value(state,  $\alpha$ ,  $\beta$ )
  if terminal_test(state) then
    return utility(state)
  end if
  v  $\leftarrow$   $\infty$ 
  for all ainactions(state) do
    v  $\leftarrow$  min(v, max_value(result_state(state, a),  $\alpha$ ,  $\beta$ ))
    if v  $\leq$   $\alpha$  then
      return v
    end if
     $\beta$   $\leftarrow$  min( $\beta$ , v)
  end for
  return v
end function

```

---