



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

29.04.2019

Bearbeitungszeitraum
Matrikelnummer, Kurs
Ausbildungsfirma
Betreuer

17.09.2018 - 29.04.2019
5388171 & 5501463, TINF16AI-BC
IBM Deutschland GmbH, Mannheim
Prof. Dr. Karl Stroetmann

Unterschrift Betreuer

Inhaltsverzeichnis

Erklärung der akademischen Aufrichtigkeit	I
Abstract	II
1 Einleitung	1
1.1 Zweck und Ziel dieser Arbeit	1
1.2 Kriterienformulierung	1
2 Theoretische Hintergründe	2
2.1 Spieltheorie	2
2.2 Evaluierungsfunktionen	11
3 Technische Grundlagen	12
3.1 Verwendete Bibliotheken	12
3.2 Architektur	12
4 Implementation	16
5 Evaluation	17
5.1 Kriterienerfüllung	17
5.2 Einordnung Intelligenz	17
Abkürzungsverzeichnis	III

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, 29.04.2019

Ort, Datum

Unterschrift Lars Dittert & Pascal Schröder



Abstract

1 Einleitung

1.1 Zweck und Ziel dieser Arbeit

1.2 Kriterienformulierung

2 Theoretische Hintergründe

Vor dem Erstellen einer Schach-KI (Künstliche Intelligenz) muss zunächst betrachtet werden, wie der Aufbau von KIs für spiel-mechanische Abläufe ist, welche Funktionen diese enthalten müssen und wie diese speziell auf das Spiel “Schach” angewandt werden können. Dazu werden zunächst Grundlagen der Spieltheorie betrachtet ehe näher auf Möglichkeiten zur Nutzung dieser im Kontext des Schachspiels eingegangen wird. Im zweiten Teil werden die verwendeten Methoden für die in dieser Arbeit beschriebene KI erläutert und die Wahl begründet.

2.1 Spieltheorie

2.1.1 Geschichte

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. Viel mehr 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 2.1.2 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 voraus geschaut 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 spielende KI, ebenfalls basierend auf den Minimax Algorithmen. Die Problematik beim Schach im Vergleich zu anderen Spielen sind die enorme Zahl an verschiedenen Zügen und Spielzuständen, die eine Evaluierung jedes einzelnen selbst bei heutiger Rechenkapazität unmöglich macht. Dies ist genauer in Kapitel 2.1.4 beschrieben.

Auf Grund dieser Komplexität des Schach Spiels sowie der begrenzten Rechenkapazitäten der damaligen Computer jedoch wurde dies auf eine Tiefe von 4 begrenzt. Dies bedeutet, dass der Computer lediglich 4 Züge vorausschaut. Zusätzlich schaut der Computer lediglich 7 verschiedene Optionen pro Zug an. Die Auswahl dieser 7 Optionen geschah mittels

simpler heuristischer Berechnungen, die versuchten im Vorab die 7 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 2.1.3 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.

2.1.2 Minimax-Algorithmus

Der Minimax Algorithmus basiert auf dem “Minimax-Theorem” von John Von 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 $\min_y \max_x G'(x, y) = \max_x G'(x, y^*) = V$. \square

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 best möglichen 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:

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

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

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

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

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

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

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

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

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

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
- *resultState(s, a)*: Der von Zustand s über Zug a erreichbare Zustand
- *terminalTest(s)*: Prüft einen Zustand darauf, ob dieser das Ende des Spiels bedeutet. Diese wird definiert als

$$terminalTest : States \rightarrow \mathbb{B}$$

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:

$$terminalStates := s \in S | terminalTest(s)$$

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

$$utility : terminalStates \times player \rightarrow \mathbb{N}$$

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 an Hand der Spielregeln und eines Ausgangszustands s_0 und einer Liste aller Züge von s $actions(s)$ alle erreichbaren Zustände $resultStates$ berechnet. Von jedem Zustand $s \in resultStates$ wird dann wiederum jeder mögliche erreichbare Zustand berechnet. Diese Schleife wird fortgeführt, bis die berechneten Zustände den Status eines Endzustandes erreichen, also $terminalTest(s) = true$ ergeben, von dem aus keine Änderungen des Zustands 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 berechenbar und evaluierbar.

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 Figur 2.2.1.1

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

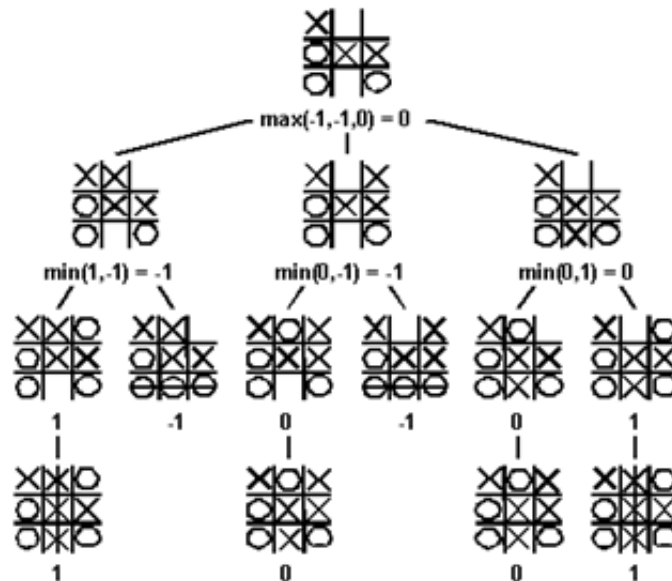


Figure 2.1.1.1 Tic Tac Toe Minimax Baum []

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 2.1.4 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 bei ?? 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. Die Implementierung eines solchen mittels drei verschiedener Funktionen kann dann wie folgt aussehen:

```

1 function minimaxDecision(state) returns state
2   for each a in actions(state) do
3     value ← minValue(resultState(state, a))
4     if value > bestValue then
5       bestValue ← value
6       bestMove ← a
7   return bestMove
8
9 function maxValue(state) returns value

```

```

10  if finished(state) then return utility(state)
11  v ← -infinity
12  for each a in actions(state) do
13    v ← max(v, minValue(resultState(state, a)))
14  return v
15
16 function minValue(state) returns value
17   if finished(state) then return utility(state)
18   v ← infinity
19   for each a in actions(state) do
20     v ← min(v, maxValue(resultState(state, a)))
21   return v

```

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 Figur 2.1.2.1 als Beispiel genommen werden:

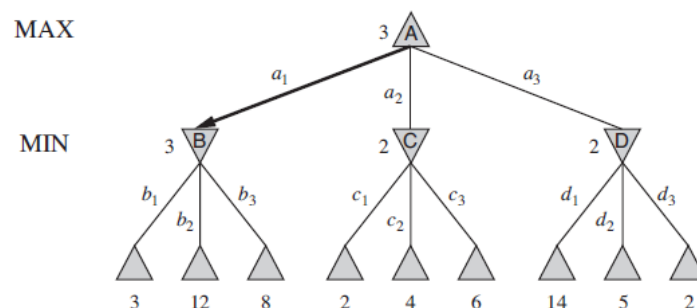


Figure 2.1.2.1 Minimax-Baum []

Dabei hat der gegebene Zustand noch eine erreichbare Tiefe von drei bis alle darauf folgenden 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 der 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 geringst mögliche Wert.

Von diesen ausgerechneten Minimas 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, zurück gegeben.

Somit ist es mit dem Minimax-Algorithmus möglich an Hand 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 zwangweise 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 m 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 2.1.3 erklärt. Das Problem, dass stets bis in die Endzustände gerechnet werden muss, wird in Kapitel 2.1.4 genauer erläutert sowie einige Lösungsansätze dargestellt.

2.1.3 Alpha-beta pruning

2.1.4 Problematik Minimax-Algorithmen komplexerer Spiele

2.2 Evaluierungsfunktionen

2.2.1 Eröffnungsstrategie

2.2.2 Figurenbewertung (+ Position)

2.2.3 Verteidigung

2.2.4 Angriff

2.2.5 Spielverlauf

2.2.6 Finishing strategy

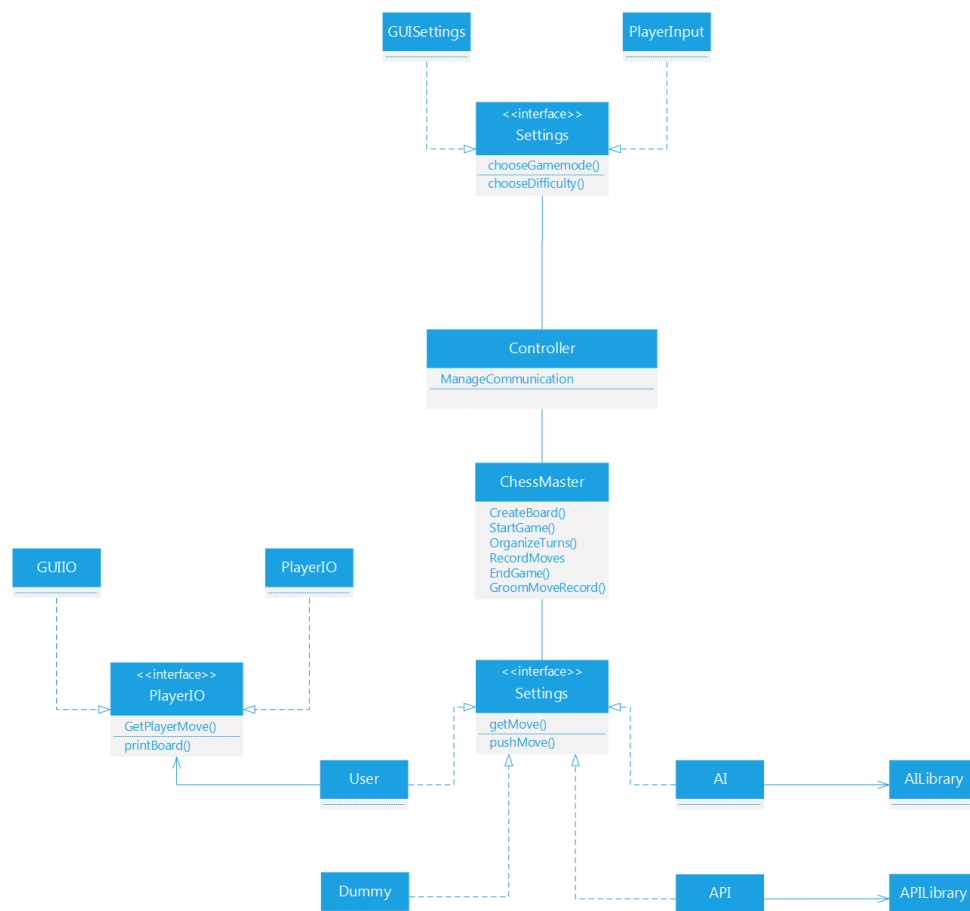
3 Technische Grundlagen

3.1 Verwendete Bibliotheken

3.2 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.



Figur 3.2.1 Klassendiagramm Architektur ChessAI

Als zentrales, Verbindungselement steht zunächst der “Controller”. 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 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

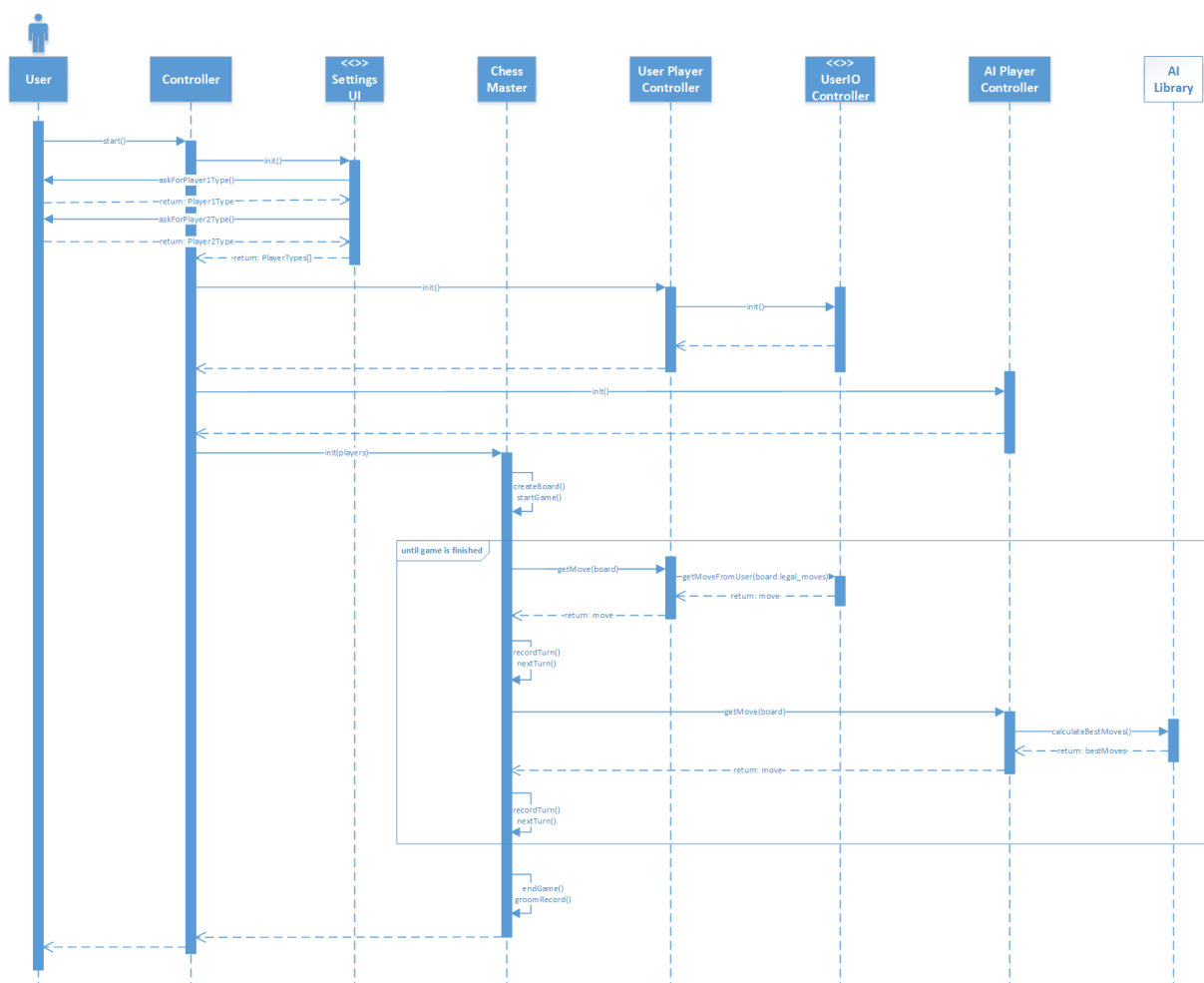
Dazu existiert eine “Player” Schnittstelle, die für jeden dieser Spieler implementiert werden muss.

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 Benutzeroberfläche 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.



Figur 3.2.2 Sequenzdiagramm Architektur ChessAI

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 nächste 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.

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 Revanche 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