



KANTONSSCHULE HOHE PROMENADE
GYMNASIUM, ZÜRICH
SCHULJAHR 2023/2024

IMPLEMENTIERUNG EINES SCHACHPROGRAMMS IN PYTHON

Tom Timur Türker

Klasse 6d

Betreuende Lehrperson: Severin Walser
Korreferent: Stefan Müller

20. Dezember 2023

Inhaltsverzeichnis

1	Einleitung	1
1.1	Schach: Altes Spiel, heute noch beliebt!	1
1.2	Motivation, Ziele und Struktur der Arbeit	2
2	Grundlagen des Schachspiels	3
3	Umsetzung des Schachprogramms	7
3.1	Grundlegendes Programmgerüst	8
3.2	Frontend	9
3.2.1	UI	9
3.2.2	Pygame Main-Schleife	9
3.2.3	Verwertung des User-Inputs	10
3.3	Backend	11
3.3.1	Verwaltung des Schachbretts	11
3.3.2	Funktionsweise eines Spielzuges	12
3.3.3	Spielzuggenerator	13
3.3.4	Spielende	14
3.4	Erweiterungen der Schach-Engine	14
3.4.1	Unterschiede zwischen Spieler und KI	14
3.4.2	Minimax-Algorithmus	15
3.4.3	Alpha-Beta-Suche	17
4	Schlussbemerkungen	19

Kapitel 1

Einleitung

1.1 Schach: Altes Spiel, heute noch beliebt!

Etwa 1500 Jahre ist es her, seitdem die erste Version des Schachs erfunden wurde. Das einfach aufgebaute Schachspiel fand direkt Beliebtheit bei der gehobeneren Schicht der Gesellschaft und breitete sich von der Umgebung Indiens in Kürze bis nach Europa aus. In den nächsten Jahren entwickelten begeisterte Schachspieler neue Taktiken und Regeln, bis im 18. Jahrhundert in Europa die ersten Ideen eines „Automaten“ aufkamen. Um 1770 wurde ein Automat namens „der Türke“ gebaut [1]. Dieser Automat war eine an einem Tisch sitzende Puppe mit orientalischem Aussehen, die zur Verwunderung aller Schachspielen konnte. Das Publikum wusste anfangs natürlich nicht, dass sich unter dem Tisch ein „professioneller“ Schachspieler befand, der durch raffinierte Technik den Arm des „Schachroboters“ steuerte.¹ Trotz des Schwindels blieb die Fantasie eines schachspielenden Automaten in den Köpfen der Menschen und sollte sich in den nächsten 200 Jahren bewahrheiten.

In den 1950er Jahren vollendete Alain Turing², ein Vorreiter auf dem Gebiet der Informatik und künstlichen Intelligenz, den ersten Schachalgorithmus namens „Turochamp“. Weil der Algorithmus jedoch auf dem damals kommerziell erhältlichen Computer Ferranti Mark I nicht implementiert werden konnte, simulierte er den Algorithmus auf einem Blattpapier [3]. Die Mächtigkeit einer Schach-Engine wird durch die Berechenbarkeit und Bewertung von möglichst vielen Spielsituationen bestimmt. Je mehr Spielsituationen ein Schachprogramm im Voraus berechnen und geeignet mit einer Gewinnwahrscheinlichkeit belegen kann, desto besser ist das Programm.

In den 1990er Jahren schritt die Schach-Engine-Entwicklung [7] mit riesigen Sprüngen voran. 1996 schaffte es die Engine von IBM's „Deep Blue“ [5] den besten Schachspieler der damaligen Zeit, den mehrmaligen Schachweltmeister Garry Kasparov, zu schlagen. 2017 wurde Alpha Zero [6] veröffentlicht, eine Engine, die Gebrauch von einem neuronalen Netzwerk machte, um sich selbst nicht nur Schach, sondern auch andere Brettspiele beizubringen. Auch wenn die Schachcommunity von den überwältigend starken Schach-Engines demoralisiert wurde, leitete die Ära der Schach-Engines keineswegs das Ende des menschlichen Schachspiels ein. Ganz im Gegenteil: Schach ist heutzutage populärer denn je, auch angeführt durch besonders junge Schachweltmeister wie Magnus Carlsen und die vermehrte Berichterstattung in den verschiedensten Medien.

¹Einige Quellen [2] führen das deutsche Verb „türken“ auf den vortäuschenden Schachroboter zurück.

²Alan Turing ist der Namensgeber für den *Turing Award*, der bedeutendsten Auszeichnung in der Informatik, sowie für den Turing-Test zum Überprüfen des Vorhandenseins von künstlicher Intelligenz.

Die meisten Schachspiele finden heutzutage über öffentlich zugängliche Webseiten wie <https://www.chess.com/> oder <https://lichess.org/> statt. Ausserdem werden die allgemein verfügbaren Schachprogramme als Lernmittel für Schachinteressierte aller Art verwendet und können als Offlinegegner gebraucht werden. Diese Programme haben noch lange nicht ihre Grenzen erreicht. Mit Neuronalen Netzwerken und Erforschungen in der Informatik, allen voran dem Aufkommen von Künstlicher Intelligenz (KI), ist das Potential der Engines unermesslich. Und wer weiss, vielleicht wird das Spiel mit den 64 Quadraten schon bald so implementiert sein, dass der Schachcomputer [4] unbesiegbar ist.

1.2 Motivation, Ziele und Struktur der Arbeit

Mir war von Anfang an klar, dass meine Maturarbeit sich im Bereich des Programmierens befinden sollte. Es stellte sich die Frage, welche Art von Programmen mich interessierten. Mein Betreuer Severin Walser machte mir ein paar Vorschläge und darunter befand sich auch die Idee der Entwicklung eines Schachprogramms. Schach als Thema passte sehr gut, da ich mich schon länger für Schach interessierte und ich somit etwas Vorwissen mitbringen konnte. Ausserdem war das Prinzip von Schach einfach genug, so dass es jeder schnell verstehen kann, und trotzdem so komplex, dass es eine Vielzahl an Wegen gibt, ein Schachprogramm aufzubauen und somit dem Programmierer einiges an Freiheit bei der Umsetzung gewährt wird.

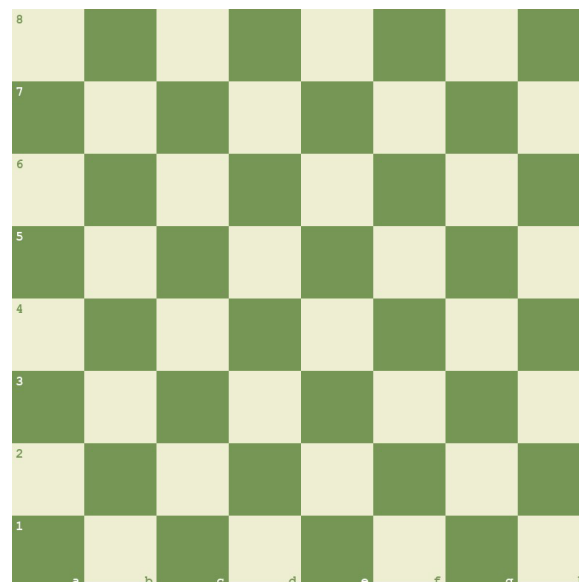
Mit dieser Maturaarbeit verfolgte ich zwei wesentliche Ziele. Einerseits sollte am Ende dieser Arbeit ein interaktives, funktionstüchtiges Schachprogramm entstehen, das alle Schachregeln befolgt. Idealerweise sollte dieses Programm einen KI-Modus anbieten, der eine Partie gegen eine KI ermöglicht. Andererseits wollte ich mich mit dieser Arbeit in die Grundlagen der Computer-Programmierung einarbeiten. Die Programmierung des Schachprogramms sollte mit Hilfe der beliebten Programmiersprache Python [8] und dem Game-Entwicklungstool Pygame [9] erfolgen. Auf beide werde ich im Nachfolgenden unter anderem eingehen. Im Vordergrund dieser schriftlichen Ausarbeitung steht die Beschreibung der implementierten Funktionalität meines Schachspielprogramms. Dabei versuche ich auch den Prozess zu skizzieren, wie das Schachprogramm entstanden ist und welche Probleme dabei zu lösen waren.

In Kapitel 2 werde ich die Grundlagen des Schachspiels zusammenfassen und später benötigte Begriffe einführen. Kapitel 3 beschreibt die verschiedenen Komponenten meines Schachprogramms und zeigt einige Implementierungsdetails auf. Dort gehe ich exemplarisch darauf ein, wie bestimmte Komponenten meines Schachprogramm programmiert wurden und zeige Schwierigkeiten auf, die mir begegneten. Kapitel 4 schliesst die Arbeit mit Bemerkungen und Erkenntnissen ab, die ich im Rahmen dieser Arbeit gewonnen habe.













Kapitel 2

Grundlagen des Schachspiels

Schach ist ein Brettspiel für zwei Personen. Das quadratische Schachbrett (engl. *chess board*) besteht aus 8x8 Feldern (engl. *squares*), die abwechselnd hell- und dunkelfarbig sind. Die Zeilen (engl. *ranks*) sind von 1 bis 8 nummeriert; die Spalten (engl. *files*) werden mit den Buchstaben a bis h versehen.¹

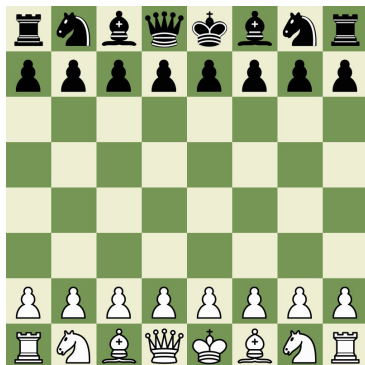


Ein Spieler nimmt die *weissen* Figuren (Spielsteine); der andere die *schwarzen*. Zu Beginn des Spiels werden jeweils 16 der folgenden Figuren an die Spieler verteilt:²

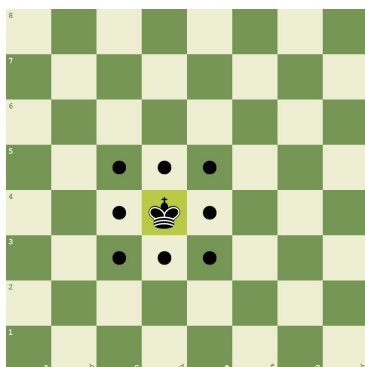
Figur	Piece (engl.)	Weiss	Schwarz	Anzahl
König	King			1
Dame	Queen			1
Läufer	Bishop			2
Springer	Knight			2
Turm	Rook			2
Bauer	Pawn			8

¹Alle in dieser Arbeit verwendeten Schachbrett-Abbildungen habe mit Hilfe von Screenshots des von mir entwickelten Schach-Frontends erstellt.

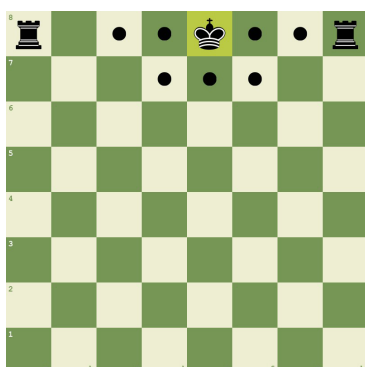
²Die englischen Bezeichnungen des Schachspiels, insbesondere die der Figuren, sind bei der Programmierung des Schachspiels wichtig. Deshalb sollte man sie kennen.



Links sehen wir die Aufstellung der Figuren am Anfang des Spiels. Der Spieler mit den weissen Figuren beginnt das Spiel. Danach wird abwechselnd gezogen. Das Ziel des Spieles ist das Mattsetzen des gegnerischen Königs, sodass dieser sich nicht mehr aus dem Schach (Angriff) des Gegners befreien kann. Alle Schachfiguren besitzen individuelle Zug- und Schlagfähigkeiten, die nachfolgend dargestellt werden.

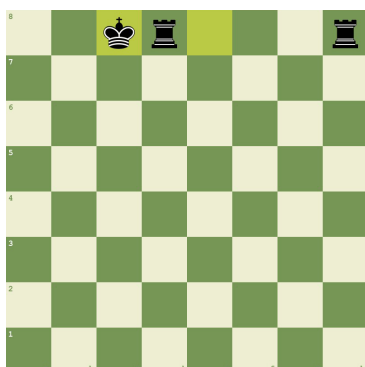


Der König zieht und schlägt diagonal, vertikal oder horizontal nur ein Feld weit, sofern dieses nicht im Schach einer gegnerischen Figur steht.

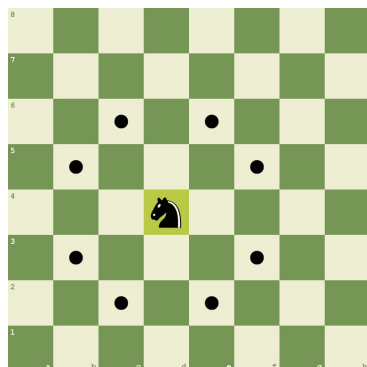


Mittels „Rochade“ kann der König zwei Felder seitlich ziehen, wenn ein Turm aus der Ecke heraus über den König auf das Feld direkt neben dem König springt. Vorausgesetzt Folgendes gilt:

1. Der König hat noch keinen Zug gemacht.
2. Der an der Rochade beteiligte Turm ebenfalls nicht.
3. Zwischen dem König und rochierendem Turm stehen keine anderen Figuren.
4. Der König steht weder im Schach noch überquert er ein im Schach stehendes Feld.

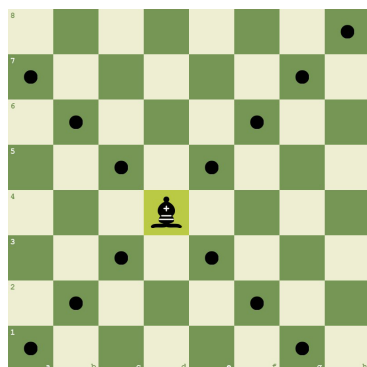


Der König *steht im Schach*, wenn er durch eine gegnerische Figur mit dem nächsten Zug geschlagen werden könnte.

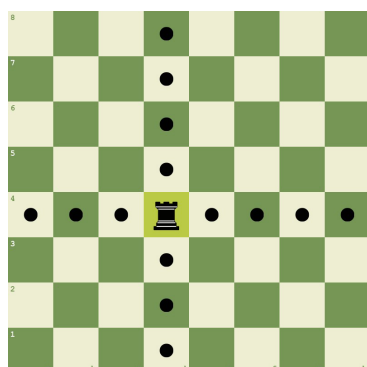


Der Springer zieht und schlägt ein Feld, das nur mit einem vertikalen oder horizontalem Zug in Kombination mit einem Diagonalfeldzug in der „gleichen“ Richtung erreicht werden kann.

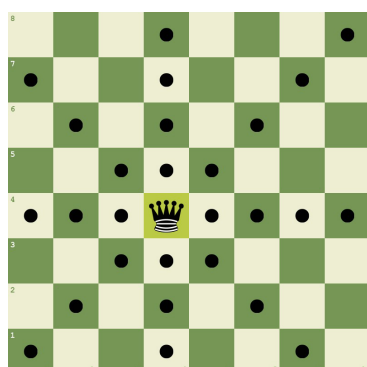
Der Springer die einzige Figur des Schachspiels, die andere Figuren überspringen kann. Diese aussergewöhnliche Funktion hat ihm auch seinen Namen in der deutschsprachigen Schachterminologie verliehen.



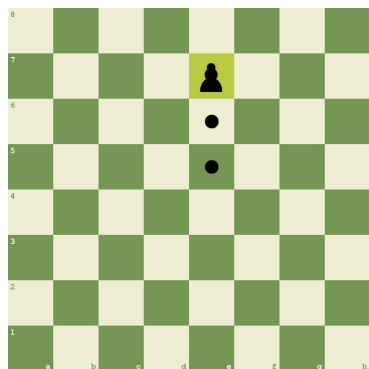
Der Läufer zieht und schlägt nur diagonal. Dabei darf er keine (eigenen oder gegnerischen) Figuren überspringen.



Der Turm zieht und schlägt vertikal oder horizontal, ohne dabei andere Figuren zu überspringen. Ausserdem kann der Turm, wie zuvor besprochen, mit dem König in einer Rochade gespielt werden.

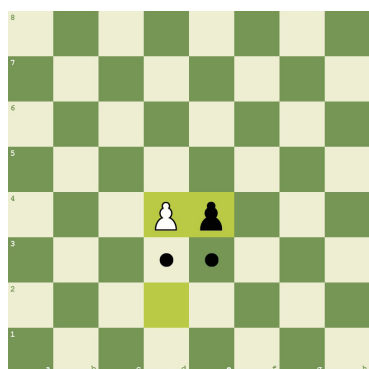


Die Dame ist die mächtigste Figur im Schachspiel und vereint die Fähigkeiten von Läufer und Turm. Wie auch der Läufer und Turm, kann die Dame über mehrere Felder hinweg gezogen werden, darf dabei aber genauso auch keine Figuren überspringen.

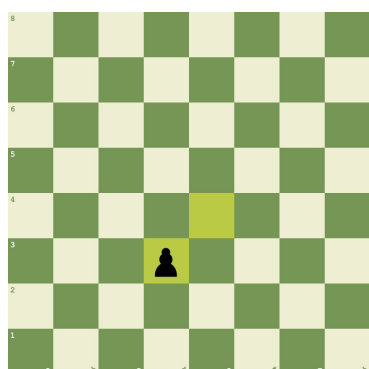


Beim Bauer wird zwischen „ziehen“ und „schlagen“ unterschieden. Der Bauer zieht ein Feld vertikal in seine Bewegungsrichtung auf ein leeres Feld oder schlägt ein Feld diagonal in seine Bewegungsrichtung, wenn auf dem Feld eine gegnerische Figur steht.

Der Bauer besitzt ausserdem noch folgende Sonderregeln:



1. Der Bauer darf bei seinem ersten Zug bis zu zwei Felder vorrücken.
2. Wenn der gegnerische Bauer Sonderregel 1 anwendet, also zwei Felder nach vorne zieht und der eigene Bauer ein Feld neben dem gegnerischen steht, darf der eigene Bauer einen Zug „im Vorbeigehen“ (*en passant*) anwenden und den gegnerischen Bauer diagonal Schlagen. *En passant* ist der einzige Zug, bei dem eine gegnerische Figur geschlagen werden kann, ohne dass die schlagende Figur danach auf dem Feld der geschlagenen Figur steht.



3. Erreicht der Bauer die gegnerische Grundzeile (weisser Bauer Rang 8 bzw. schwarzer Bauer Rang 1), wird er nach beliebiger Wahl des Spielers in eine Dame, Läufer, Turm oder Springer umgewandelt.

Im Folgenden bezeichne ich alle Spielzüge, die unter Beachtung aller Spielregeln möglich sind, als *valide*.

Spielende

Eine Schachpartie endet wie folgt [10]:

1. *Schachmatt* (engl. *checkmate*): Der Spieler, dessen König im Schach steht und sich nicht mehr aus dem Schach befreien kann, verliert das Spiel.
2. *Remis* (engl. *draw*): Unter folgenden Bedingungen endet die Partie unentschieden:
 - *Patt* (engl. *stalemate*): Der am Zug befindliche Spieler kann keinen validen Zug machen, wobei sein König nicht im Schach steht.
 - *Unzureichendes Material*: Es gibt nicht genügend Figuren auf dem Brett, um ein Schachmatt zu erzwingen.
 - *Wiederholung*: Ein Spieler beansprucht ein Remis, wenn dieselbe Stellung zum dritten Mal auf dem Brett erscheint.
 - *50-Züge-Regel*: Nach fünfzig aufeinanderfolgenden Zügen endet die Partie, wenn weder ein Bauer gezogen noch eine Figur geschlagen wurde.

Kapitel 3

Umsetzung des Schachprogramms

Schachprogramme bestehen in der Regel aus zwei Teilen:

- Das *Frontend* ist zuständig für die (graphische) Darstellung des Schachspiels sowie für die Realisierung der Benutzerinteraktion.
- Die *Engine* bzw. das *Backend* berechnet die validen Spielzüge, zieht die Figuren, und überprüft das Schachbrett auf ein mögliches Spielende.

Heutzutage gibt es eine Reihe von leistungsstarken Schachprogrammen, die in verschiedensten Programmiersprachen geschrieben wurden. Ziel meiner Arbeit war es nicht diese existierenden Schachprogramme nachzuimplementieren oder gar zu verbessern, was mit dem Wissensstand eines Programmieranfängers ohnehin ein unmögliches Unterfangen gewesen wäre. Ich wollte mich in das Gebiet der Computer-Programmierung einarbeiten und dabei ein Schachprogramm entwickeln. Als Programmiersprache habe ich mich für Python entschieden, eine einfach verständliche Programmiersprache. Sinn dieser Arbeit ist eigenständig das Frontend und Backend zu entwickeln und bei Problemen einen persönlichen Lösungsansatz zu finden.

Im Hauptteil meiner Arbeit möchte ich darlegen, wie ich mein Schachprogramm in Python programmiert habe, indem ich auf die folgenden wesentlichen Punkte eingehe:

1. Gestaltung des Schach-Frontends mittels Pygame:

- Implementierung der UI
- Pygame Main-Schleife
- Verwertung des User-Inputs

2. Zentrale Aufgaben der Schach-Engine:

- Verwaltung des Schachbrettes
- Funktionsweise eines Spielzuges
- Spielzuggenerator
- Spielende

3. Erweiterung und Optimierung der Schach-Engine:

- Unterschiede zwischen Spieler und KI
- Minimax-Algorithmus
- Minimax-Erweiterung: Alpha-Beta-Suche

3.1 Grundlegendes Programmgerüst

Ein durchstrukturierter Aufbau einer Arbeit ist beim Programmieren essenziell. Wenn die Grundlage instabil ist, wird das fertige Programm mit Sicherheit auch instabil sein. Deswegen war mir von Anfang an bewusst, dass die ersten Bausteine, die in meinem Programm gelegt wurden, die wichtigsten werden würden. Umso schwerer ist es mit wenig Erfahrung solch essenzielle Entscheidungen zu treffen. Ich war mir sicher, dass der erste Versuch mitnichten der Letzte sein würde. Somit erstellte ich „Chess Tryout“, meinen ersten Versuch. Ich kann mich noch erinnern als ich das leere Programm, ohne eine einzige Zeile Code, angestartet habe und mir dachte: „Was nun?“. Sich einfach hineinzustürzen, um zufällig den Masterplan zu finden, schien keine Lösung zu sein.

Also ging ich einen Schritt zurück und befragte das Internet. Ich stiess auf verschiedenste Ansätze und Ideen, wie man ein Schachprogramm aufbauen sollte. Mit der Zeit wurde mir klarer, was allgemein als wichtig angesehen wird, und was ich in meinem Programm verwirklichen wollte. „Chess Tryout“ wurde zu „Chess Test“. Es war ein direkter Unterschied in der Struktur des Programms zu sehen. Auch meine Zuversicht, ein besseres Schachprogramm bauen zu können, wuchs. Mir war nun klar, was meine Python-Klassen umsetzen sollten und in welcher Hinsicht ich sie brauchen würde.

Drei grosse Klassen bilden das Fundament für mein Python-Programm:

```
class Board
class Square
class Piece
```

Diese Klassen setzen die Ebenen um, in denen die Hauptfunktionalität des Schachspiels geschrieben wurde. Die Idee hinter diesen Klassen ist einfach. Stellen Sie sich als Vergleich Google Earth vor. Wenn man Google Earth öffnet, wird die Erde als Ganzes präsentiert. Nun bedarf es den User von der Zoom-Funktion Gebrauch zu machen. Beim reinzoomen werden zuerst die Landesgrenzen sichtbar, weiter die Grenzen innerhalb des Landes z.B. die Kantonsgrenzen der Schweiz. Man könnte nun die Welt in verschiedene Ebenen auf-fassen. Die oberste Ebene wäre die Erde selbst, danach würden Länder kommen, gefolgt von Grenzen innerhalb der Länder, der Kantone.

Versucht man sich nun das Schachspiel vor Augen zu führen, kommt man schnell zum Schluss, auch dieses in verschiedene Ebenen einzuteilen. Die oberste Ebene ist das Schachbrett an sich. Auf diesem Brett basiert das ganze Spiel. Ohne Brett würde alles Weitere auf dem Brett nicht existieren und man hätte kein Schachspiel mehr. Die nächste Ebene bilden die Felder. Die 64 Felder sind dafür gedacht, Ordnung auf das Brett zu bringen. Auf der obersten Ebene kommen die Schachfiguren ins Spiel. Ein Brett mit Feldern bringen einem nichts, ohne zu wissen, auf welchem der 64 Felder sich eine Figur befindet.

Zuletzt stellt sich noch die Frage: Wieso überhaupt Gebrauch von Ebenen machen? Theoretisch könnte man das Programm auch auf einer einzigen Ebene verfassen. Der Nachteil wäre eine mangelhafte Struktur. Die verschiedenen Dateien wären nicht mehr geordnet und man würde schnell die Übersicht verlieren. Deswegen die verschiedenen Ebenen. Als Programmierer weiss man direkt, wo man sich aufhält, wo man spezifischen Code finden würde oder wo man eine neue Funktion integrieren sollte. Dieser Aufbau war wichtig für ein schnelles und effektives Arbeiten am Schachprogramm.

3.2 Frontend

3.2.1 UI

Um ein Programm mit Pygame zu entwickeln, muss sowohl das Backend als auch das Frontend, die sogenannte UI (*User Interface*), berücksichtigt werden. In diesem Abschnitt möchte ich die Ziele, die ich mir für meine UI gesetzt hatte, beschreiben:

- **Brett:** Das Schachbrett, mit zwischen beige und dunkelgrün alternierenden Feldern [11], wird auf dem Pygame Fenster abgebildet.
- **Figuren:** Die Schachfiguren werden, mit einer auf die Felder abgestimmten Grösse, auf das Schachbrett platziert.
- **Verfeinerung des Bretts:** Die Zeilen werden von a bis h alphabetisiert und die Spalten von 1 bis 8 nummeriert.
- **Geklickte Figur:** Die vom Spieler angeklickte Figur wird in einem grellen gelb hervorgehoben.
- **Valide Züge der geklickten Figur:** Ausserdem werden Felder, zu denen die gerade angeklickte Figur ziehen kann, durch einen kleinen Kreis auf dem Feld gekennzeichnet. Züge, die zu dem Schlagen einer gegnerischen Figur führen, werden mit einem Ring statt einem Kreis vorgehoben
- **Letzter gespielter Zug:** Ein grelles gelb hebt den letzten Spielzug hervor.
- **Spieltöne:** Bei Vollendung eines Zuges wird ein Ton abgespielt. Der Ton variiert beim Schlagen einer Figur.
- **Spielmodi:** Links neben dem Spielbrett werden die Schaltflächen der vier Spielmodi angezeigt.

In den folgenden Abschnitten zeige ich exemplarisch mit der Darstellung des Schachbrettes sowie der angeklickten Figur, wie ich dies mit Pygame umgesetzt habe.

3.2.2 Pygame Main-Schleife

Ich wollte von Anfang an mein Python-Programm interaktiv gestalten. Das Schachbrett sollte nach jedem Spielzug aktualisiert werden und auf Klicks sowie Mausbewegungen reagieren. Um ein solches Programm zu erstellen, sind Game-Entwicklungstools wie Unity oder Unreal Engine unerlässlich. Game-Entwicklungstools unterstützen den Programmierer durch die Bereitstellung vieler Komponenten, ein Computerspiel von Grund auf zu bauen. Meine Entscheidung fiel auf Pygame. Pygame nimmt meinen Code und spielt ihn auf einem interaktiven Fenster ab. Ausserdem kann Pygame jede vom Benutzer getätigte Aktion aufnehmen und dementsprechend den Durchlauf der Main-Schleife (Hauptprogrammcode) ändern.

Die Main-Schleife wird so lange durchlaufen, bis der Benutzer sich dazu entscheidet, das Programm zu beenden. Zuerst müssen die Parameter des von Pygame projizierten Bildschirms `screen_x` und `screen_y` festgelegt werden. Mein Monitor läuft auf Full HD (1920 x 1080 Pixel), was ich auf die Bildschirmvariablen übertragen habe. Beim Starten des Programms wird nun Pygame einen leeren Bildschirm anzeigen, der aus 1920 Pixel

in der x-Achse und 1080 Pixel in der y-Achse besteht. Auf diesem leeren Bildschirm kann ich das Schachbrett, die Figuren und die Buttons, die für das Auswählen der Spielmodi verantwortlich sind, darstellen.

```
def main():
    screen = pygame.display.set_mode((screen_x, screen_y))
    clock = pygame.time.Clock()
    while True:
        print_board(screen)    # projiziert das Schachbrett
        pygame.display.flip()  # erneuert das angezeigte Bild
        clock.tick(60)         # Durchläufe der Schleife pro Sekunde
```

Den Durchlauf der Main-Schleife kann man sich wie einen Film vorstellen. Ein Film ist eine Aneinanderreihung von Bildern. Je höher die Wiederholungsrate ist, desto mehr Bilder werden pro Sekunde aneinandergereiht und desto flüssiger nehmen wir den Film wahr. Die Anweisung `clock.tick(60)` setzt die Wiederholungsrate des Programms auf 60. Die Main-Schleife wird also 60-mal pro Sekunde durchlaufen und anfallende Änderungen durch die Anweisung `pygame.display.flip()` vorgenommen. Eine höhere Framerate führt zu flüssigeren Bewegungen, aber nimmt mehr Leistung des Rechners in Anspruch. Ich habe mich für eine Framerate von 60 entschieden; die Bewegungen fühlen sich flüssig an, während der Anspruch an Leistung ertragbar ist.

```
def main():
    screen = pygame.display.set_mode((screen_x, screen_y))
    clock = pygame.time.Clock()
    while True:
        screen.fill((0, 0, 0)) # übermalt das Bild in schwarz
        print_board(screen)
        pygame.display.flip()
        clock.tick(60)
```

Die Anweisung `screen.fill(0, 0, 0)` färbt den Bildschirm komplett schwarz. Die Zahlen in der Klammer stehen für die RGB-Werte einer Farbe. (0, 0, 0) beinhaltet 0 Rot, 0 Grün und 0 Blau, was in RGB schwarz bedeutet. Stellen Sie sich das Schwarzfärben des Bildschirms als Radiergummi vor. Ohne diese Anweisung würde man die vorherigen Iterationen des auf den Bildschirm gedruckten Schachbrettes erkennen, was bei 60 gedruckten Schachbrettern pro Sekunde schnell unübersichtlich werden kann. Indem wir den Bildschirm bei jedem Schleifendurchlauf komplett schwarz färben, haben wir einen sauberen Bildschirm, auf dem wir die neueste Version des Schachbrettes drucken und sichtbar machen können. Das Kommando `pygame.display.flip()` erneuert das Bild in diesem Fall mit jedem Schleifendurchlauf. Deshalb wird man den komplett schwarzen Bildschirm auch nie sehen können; weil nur der Stand am Schluss der Schleife, also immer das Schachbrett auf dem schwarzen Bildschirm, angezeigt wird.

3.2.3 Verwertung des User-Inputs

Die bisherige Main-Schleife kann die ganze UI anzeigen und bei Änderungen des Schachbretts immer die neueste Version darstellen. Jedoch ist dem Benutzer eine Beteiligung am Spiel bislang verwehrt, d.h., ein Spieler kann noch nicht mit dem Programm interagieren. Das Programm wurde nun so erweitert, dass User-Input entgegengenommen und im

Spiel verarbeitet werden kann. Dafür verwendete ich von Pygame bereitgestellte Funktionen, die es erlauben, bei bestimmten Nutzereingaben mit entsprechenden Aktionen zu reagieren. Der folgende Code zum Beispiel illustriert das Highlighten der angeklickten Spielfigur:

```
def main():
    screen = pygame.display.set_mode((screen_x, screen_y))
    clock = pygame.time.Clock()
    while True:
        screen.fill((0, 0, 0))
        print_board(screen)
        for event in pygame.event.get():
            if event.type == pygame.MOUSEBUTTONDOWN:
                highlight_clicked_piece(screen, piece)
        pygame.display.flip()
        clock.tick(60)
```

Ich nutze also sogenannte Pygame *Events*, um auf bestimmte Ereignisse, in diesem Fall den User-Input „Maus-Taste Drücken“, zu reagieren. Mit Hilfe der allgemeinen Abfrage `if event.type == pygame.EVENT` reagiert man auf ein spezifisches `event`. So konnte ich mit einer erweiterten Main-Schleife und den verschiedenen Events, die Pygame unterstützt, sämtliche in Abschnitt 3.2.1 genannten UI-Ziele analog umsetzen.

3.3 Backend

3.3.1 Verwaltung des Schachbretts

Wie ich das Schachbrett in Code verwalten wollte, war wohl die bedeutsamste Entscheidung, die ich für mein Programm treffen musste. Grundsätzlich gibt es drei Ansätze, zwischen denen ich mich entscheiden musste:

1. Bitboards (Bitvektoren) der Länge 64
2. Liste mit 64 Feldern
3. Listen mit jeweils 8 Feldern

Ansatz 1, die Bitboard-Verwaltung, ist die effizienteste Variante. Heutzutage ist sie praktisch in jeder starken Schach-Engine vertreten. Ich hatte mich gegen diesen Ansatz entschieden, weil er mit Abstand der komplizierteste unter den dreien ist. Das komplexe Thema Bitboard-Verwaltung hätte ohne das nötige Informatikvorwissen eine ziemlich lange Einarbeitungsphase benötigt. Ich wollte jedoch sichergehen, dass ich am Ende der Arbeit ein lauffähiges Programm habe, welches ich auch vorführen kann.

Die anderen beiden Ansätze sind listenbasiert. Ein Listenfeld vertritt eines der 64 Felder des Schachbretts. Der Unterschied zwischen den beiden Ansätzen liegt in der Anzahl der verwendeten Listen. Ansatz 2 nutzt ein einzige Liste mit 64 Feldern, während Ansatz 3 von acht Listen mit je acht Feldern Gebrauch macht. Weil Python mit einem einzelnen Liste schneller arbeitet, mag dieser Ansatz aus Sicht der Berechnungen effizienter sein. Jedoch kann man bei Ansatz 3 direkt ablesen, auf welchem Feld sich eine Figur befindet. Das Feld auf Zeile 5, Spalte 4 zum Beispiel findet man erheblich schneller auf

dem Schachbrett als das Feld auf der Position 36, welches zunächst eine Übersetzung der Zeilen/Spalten-Kombination in die zugehörige Feldposition benötigt. Diese vereinfachte Leseart und der Glaube, dass der Effizienz-Unterschied zwischen Ansatz 2 und 3 im Spiel später nicht ausschlaggebend sein würde, führten dazu, dass ich mich für Ansatz 3 zur Verwaltung des Schachbretts entschied. In Python werden die Felder demnach wie folgt als Liste von Listen verwaltet:

```
squares = [[Square(0, 1), Square(0, 2)... Square(0, 7)]
            [Square(1, 1), Square(1, 2)... Square(1, 7)]
            ...
            [Square(7, 1), Square(7, 2)... Square(7, 7)]]
```

3.3.2 Funktionsweise eines Spielzuges

Die Implementierung der Spielzüge ist zentral für jedes Schachprogramm. Hier müssen alle in Kapitel 2 aufgelisteten Spielregeln beachtet werden. In Python habe ich hierzu eine Funktion namens `move` programmiert, welche die folgenden drei Parameter entgegennimmt, um einen Spielzug durchzuführen:

```
initial_square # Ausgangsfeld der Spielfigur
final_square   # Zielfeld der Spielfigur
promotion_piece # Eingetauschte Figur bei der Umwandlung
```

Diese Funktion überführt die Spielfigur, die auf dem Feld `initial_square` ist auf das Feld `final_square`. Der dritte Parameter namens `promotion_piece` kommt bei der Umwandlungsregel zum Tragen, wenn die gezogene Figur ein Bauer ist, die auf der gegnerischen Grundzeile landet. Dann wird der Bauer durch das `promotion_piece` ersetzt. Stand zuvor eine gegnerische Figur auf dem Zielfeld, ist sie mit dem „Überschreiben des Feldes“ geschlagen worden. Der vereinfachte Code sah anfangs so aus:

```
move = Move(initial_square, final_square, promotion_piece)

def move(self, piece, move):
    squares[move.initial_square.rank][move.initial_square.file].piece
    = None
    squares[move.final_square.rank][move.final_square.file].piece
    = piece
```

Die Funktion war also zunächst recht überschaubar. Doch mit der Berücksichtigung jeder weiteren Spielregel wurde sie immer komplexer und damit schwieriger zu überblicken, was wiederum zu einer hohen Fehleranfälligkeit führte. Die anfangs kleinwirkende Funktion enthüllte ihre wahre Grösse vor allem mit der Implementation der Rochade. Bei jedem Zug müssen die Parameter der Rochade neu beurteilt werden. Insbesondere, wenn man die inverse Funktion `unmake_move` hinzufügt, die das Schachbrett um den zuletzt getätigten Zug zurücksetzt. Dann muss sich das Programm auch noch den Zug, in welchem die Parameter der Rochade umgeändert wurden, merken.

Auf die beiden Funktionen `move` und `unmake_move` musste ich im Verlauf des Programmierens öfters zurückkommen, was meiner Fehleinschätzung der notwendigen Implementierungen geschuldet war.

3.3.3 Spielzuggenerator

Die Zugmöglichkeiten einer Figur lassen sich aus ihren Bewegungsrichtungen und den Schachregeln (siehe Kapitel 2) berechnen. Turm, Läufer und Königin können sich unter bestimmten Bedingungen über mehrere Felder in eine Richtung bewegen. Der grundlegende Code zur Realisierung der Spielzüge für diese drei Figuren ist daher praktisch derselbe. Nur ihre Bewegungsarten (diagonal und/oder vertikal) lassen sich voneinander unterscheiden. Mit dieser Logik benötigte ich nur eine Funktion, um die Spielzüge von Turm, Läufer und Königin zu berechnen. Für die Zugmöglichkeiten der drei weiteren Figuren (König, Springer und Bauer) fand ich keine bedeutsamen Ähnlichkeiten, um Code zu sparen, weswegen ich sie einzeln berechnet habe.

Die wirklichen Schwierigkeiten bei der Zugberechnung treten erst mit der Betrachtung der Regeln des Königs auf. Wenn der eigene König angegriffen wird, also im Schach steht, muss der Spielzuggenerator alle Züge eliminieren, die den König nicht aus dem Schach befördern können. Um zu überprüfen, ob der eigene König angegriffen wird, müssen zuerst alle gegnerischen Züge ausgerechnet werden. Dafür habe ich die Funktion `calculate_enemy_attacking_moves` entwickelt. Diese Funktion speist zwei Listen aus, `enemy_attacking_squares` und `enemy_checking_squares`[12].

`enemy_attacking_squares` ist wie `squares` eine zweidimensionale Liste, dessen Felder mit Nullen und Einsen anstatt `Square`-Elementen gefüllt sind. Die Eins repräsentiert ein Feld, das von einer gegnerischen Figur angegriffen werden kann; die Null ein von den Gegnern nicht-attackiertes Feld. Diese zweidimensionale Liste wird für die Bestimmung der Zugmöglichkeiten des Königs gebraucht. Wenn das Feld, auf das der König ziehen möchte mit einer Null gekennzeichnet ist, ist der Zug valide und kann somit gespielt werden. Bei Feldern mit einer Eins werden die Züge verhindert, da sie invalide sind.

`enemy_checking_squares` ist eine Liste, welche Felder speichert, von denen ein Angriff auf den eigenen König ausgeht und ihn somit in Schach setzt. Meistens ist dies nur einem einzigen Feld, nämlich das, auf dem die angreifende Figur steht. Im Fall einer Königin, eines Läufers oder Turms muss jedoch der ganze Angriffsweg in Betracht genommen werden, weil man dem Schach nicht nur durch das Schlagen der angreifenden Figur entkommen kann, sondern auch durch das Reinstellen einer eigenen Figur in den Angriffsweg des Gegners. Die reingestellte Figur nimmt hierbei die Funktion einer Blockade ein und kann somit auch den König aus dem Schach ausbrechen lassen. Wenn also der König im Schach steht, werden nur diejenigen Züge als valide akzeptiert, die sich auf eines der `enemy_checking_squares` stellen. Tritt ein Fall ein, in dem mehrere gegnerische Figuren den König in Schach versetzen, werden zeitsparend nur die Züge des eigenen Königs berechnet, weil unter keinen Umständen eine andere Figur als der König selbst ihn von einem Doppelangriff schützen kann.

Zuletzt werden noch die `pinned_squares` der Figuren berechnet, denen das Brechen des „pins“, d.h., das Wegziehen einer Figur, die ursprünglich den König vor dem Schach bewahrt hat, verwehrt wird. Zusammengesetzt sah meine Funktion zur Berechnung der validen Züge folgenderweise aus:

```
def get_valid_moves(color):
    calculate_enemy_attacking_moves(color)
    if enemy_checking_squares:
        return in_check_valid_moves(color)
    else:
        return no_check_valid_moves(color)
```

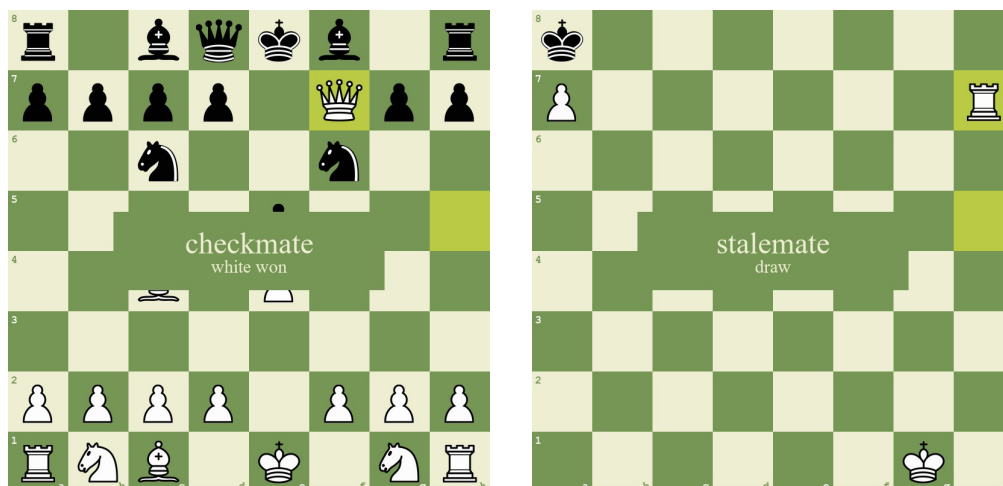

3.3.4 Spielende

Die fünf Möglichkeiten, ein Schachspiel zu beenden, wurden in Kapitel 2 besprochen: Schachmatt, Patt, Unzureichendes Material, Dreifache Wiederholung und 50-Züge-Regel. Bei der Programmierung habe ich mich zunächst auf Schachmatt und Patt fokussiert, weil sie die üblichsten und bekanntesten Arten sind. Patt ist, wenn der Spieler, der am Zug ist, keinen validen Zug spielen kann. Schachmatt ist wie Patt, nur dass der Spieler noch dazu im Schach steht.

Mit der Funktion des „Move Generator“ wurde das Fundament fürs Prüfen von Schachmatt und Patt gelegt. Mit dem Einbau weniger Strukturen konnte der Prüfer einwandfrei ein Schachmatt und Patt erkennen. Wichtig anzumerken ist, zuerst auf Schachmatt und danach erst auf Patt zu prüfen. Anderenfalls würde ein Schachmatt fälschlicherweise als Patt erkannt werden.

Die drei Regeln unzureichendes Material, dreifache Wiederholung und 50-Züge-Regel wurden im Schach eingeführt, um sinnloses Umherschieben der Figuren vorzubeugen, wenn die Partie ohnehin auf ein Remis hinausläuft. Diese drei Formen des Spielendes habe ich auch vollständig umgesetzt, auch wenn sie selten vorkommen und aufwendiges Programmieren beanspruchten.

Die nachfolgende Abbildung zeigt zwei Möglichkeiten eines Spielendes, abgebildet auf dem Frontend meines Programms:



3.4 Erweiterungen der Schach-Engine

3.4.1 Unterschiede zwischen Spieler und KI

In der ersten Phase dieser Arbeit war meine Schach-Engine auf zwei menschliche Spieler ausgelegt, die gegeneinander Schachspielen. Jede Funktion ging davon aus, dass menschliche Spieler sie benutzen würden. Unter Anderem fand keine Bewertung der möglichen Spielzüge und somit automatisch eine Auswahl des nächsten Spielzuges statt. Schliesslich ist dies die Aufgabe des Schachspielers! Als ich eine KI zu meiner Schach-Engine hinzufügen wollte, glaubte ich, nur einen menschlichen Schachspieler simulieren zu müssen, damit das Schachprogramm weiterhin einwandfrei läuft. Wie falsch ich doch lag!

Als ich die erste Erweiterung der Schach-Engine vornahm, bei der der Computer nur zufällig auswählte Züge spielte, kam mir eine Flut von Fehler-Nachrichten entgegen.

Der Code schien zwar grundlegend korrekt zu sein, jedoch haperte es an unscheinbaren Variablen, die bei der Engine fehlten oder zu viel waren. Zum Beispiel war meine Funktion für die Bewegung der Bauern auf einen menschlichen Spieler ausgerichtet, der logisch ableiten kann, dass er auf dem Bildschirm die Figur anklicken sollte, zu die er seinen Bauern befördern möchte. Bevor ich an einer fortgeschritteneren Engine arbeiten konnte, musste ich mich darum kümmern, praktisch jede grosse Funktion so umzuschreiben, dass sie KI-tüchtig war, also dass die Auswahl der zu spielenden Figuren selbständig von der Engine getroffen werden konnte. Wie erwähnt, im ersten Schritt mit Hilfe einer zufälligen Auswahl der gespielten Züge. In weiteren Verfeinerungsschritten habe ich zunächst den Minimax-Algorithmus implementiert und diesen danach mit Hilfe der Alpha-Beta-Suche ausgebaut. Auf diese beiden Ansätze gehe ich in den nächsten Abschnitten ein.

3.4.2 Minimax-Algorithmus

Der Minimax-Algorithmus [13, 14] basiert auf einer rekursiven (sich immer wieder selbst aufrufenden) Bewertungsfunktion, die den nächstbesten Zug berechnet.

Ein Durchlauf des Algorithmus spiegelt einen theoretischen Zug wider, was durch das rekursive Aufrufen zu einer Möglichkeit führt das Schachspiel mehrere Schachzüge im Voraus zu simulieren. Der Parameter `depth` bestimmt hierbei die Anzahl Halbzüge, die der Minimax-Algorithmus simuliert. Weil der Minimax-Algorithmus für alle möglichen Züge die möglichen Antworten ausrechnet, nimmt die Anzahl der auszurechnenden Züge mit der `depth` exponentiell zu. Aus diesem Grund kommt der Minimax-Algorithmus schnell an seine Grenzen, weshalb man bedachtsam den Parameter `depth` bestimmen sollte. Ist dieser Parameter zu hoch eingestellt, wird der Algorithmus zwar mehr Halbzüge simulieren, und dementsprechend eine akkuratere Bewertung des besten Spielzugs liefern können, jedoch aber auch einiges mehr an Laufzeit benötigen.

Wenn der Minimax-Algorithmus an der Spitze des Bewertungsbaumes angekommen ist, also alle möglichen Halbzüge bewertet hat, liefert er die beste Bewertung, die er finden konnte und den Zug, der zu dieser Bewertung führen würde.

Wichtig anzumerken ist, dass der Algorithmus davon ausgeht, dass der Gegner immer den besten Zug, nach seiner Bewertungsfunktion zumindest, spielen wird. Die Bewertungsfunktion meines Minimax-Algorithmus ist einfach gehalten. Jeder Figur wird im Voraus ein Wert zugeteilt. Gegnerische Figuren haben die gleichen Werte wie eigene, nur dass der Wert negativ ist. Die Bewertungsfunktion addiert alle Werte zusammen und erhält somit den Schlusswert, der in den Minimax-Algorithmus zurückgespeist wird. Anhand der Bewertungsfunktion kann man erkennen, dass der ausgespeiste Wert sich nur ändert, wenn eine Figur geschlagen wird. Die Positionen der Schachfiguren werden nicht in Betracht gezogen, was fatal für den Verlaufs des Spiels sein kann. In Schach kann man sich auch durch die richtige Anordnung eigener Figuren in eine vorteilhafte Position versetzen, ohne überhaupt eine einzige gegnerische Figur geschlagen zu haben. Diesen Aspekt zieht meine Bewertungsfunktion leider nicht in Betracht, was der Engine einen dramatischen Nachteil gegenüber anderen Schachspielern gibt, die Wert auch auf die eigene Stellung legen.

Zusammengefasst sieht die Funktion Minimax wie folgt aus:

```
def minimax(depth, maximizing_player):
    if depth == 0 or game_over():
        evaluate_position(color)
        return evaluation

    if maximizing_player:
        max_eval = -math.inf
        for move in valid_moves:

            move(piece, move)
            evaluation = minimax(depth - 1, alpha, beta, False)
            unmake_move(piece, move)

            max_eval = max(max_eval, evaluation[0])
        return max_eval
    else:
        min_eval = math.inf
        for move in valid_moves:

            minimax_move(piece, move)
            evaluation = minimax_ascended(depth - 1, alpha, beta, True)
            unmake_move(piece, move)

            min_eval = min(min_eval, evaluation[0])
        return min_eval
```

Zuerst rechnet der Minimax-Algorithmus aus, ob er sich bereits auf `depth == 0` befindet oder das Spiel zu Ende ist. Wenn ja, bewertet er die Stellung und kehrt zurück. Der restliche Teil des Minimax-Algorithmus wird durch eine „if-then-else“-Anweisung halbiert. Die zwei Hälften unterscheiden sich durch die Variable `maximizing_player`. Der ausgehende Spieler ist der `maximizing_player` und die Opposition der `minimizing_player`. Daher kommt auch der Name „Minimax“. Bei dem maximierenden Spieler wird der Zug mit dem höchsten erhaltenen Wert ausgesucht, wobei beim minimierenden Spieler der Zug mit dem kleinsten Wert genommen wird. Der auserwählte Zug des minimierenden Spieler ist, wie auch beim maximierenden Spieler, der bestmögliche Zug in der Stellung. Weil jedoch die Bewertungsfunktion gegnerische Stellungen mit einem negativen Vorzeichen bewertet, muss bei dem minimierenden Spieler die Bewertung mit dem kleinsten Wert genommen werden. Die Stellungsbewertungen werden erst vorgenommen, wenn die `depth == 0`, also der Baum des Minimax-Algorithmus vollständig gezeichnet wurde. In `for move in valid_moves` und `move(piece, move)` wird ein Zug gespielt und mit `evaluation = minimax(depth - 1, alpha, beta, False)` der Minimax-Algorithmus rekursiv wiederaufgerufen, nur dass sich der `maximizing_player` ändert. Somit ist die andere Seite am Zug und rechnet wieder alle Zugmöglichkeiten aus, betätigt `evaluation = minimax(depth - 1, alpha, beta, False)`, wodurch wieder die ursprüngliche Seite am Zug ist und alle Zugmöglichkeiten wieder ausrechnet. Der Baum wird somit exponentiell grösser, und dieses Wachstum würde weiterhin fortgesetzt werden, wenn Algorithmus nicht mit einer Variable(`depth`) eingegrenzt wäre. Bei jedem Aufruf von `evaluation`

`= minimax(depth - 1, alpha, beta, False)` wird die `depth` um Eins verringert, bis schliesslich das Ende des Baumes bei `depth == 0` erreicht wird. Dann steigt die Bewertungsfunktion `evaluate_position(color)` ein und klettert den Baum mit Hilfe von `unmake_move(piece, move)` wieder nach oben. Schlussendlich erreicht die Bewertungsfunktion die ursprüngliche Stellung und hat den besten folgenden Zug bestimmt, welchen die Engine dann final auf dem Schachbrett spielt.

3.4.3 Alpha-Beta-Suche

Obwohl der Minimax Algorithmus langsam ist und mit einfach gehaltener Bewertungsfunktion kaum erfahrene Gegner im Schach besiegen wird, hat er einiges an Verbesserungspotential, was ihn schlussendlich zu einer beliebter Option in Schach-Engines macht. Hierbei hat man zwei mögliche Ansätze. Entweder man erhöht die Genauigkeit der Bewertungsfunktion oder man versucht die Effizienz des Suchalgorithmus zu steigern.

Die Alpha-Beta-Suche (engl. *Alpha-Beta Pruning*) [14] ist eine Technik, mit relativ wenig Implementierungsaufwand die Laufzeit des Suchalgorithmus verbessert. Die Idee hinter der Alpha-Beta-Suche ist, von bereits ausgerechneten Bewertungen Gebrauch zu machen und ganze Äste des Bewertungsbaumes zu „prunen“, d.h., bei der Suche abzuschneiden, um damit den Suchraum zu verkleinern. Wenn der Algorithmus durch die Werte `alpha` und `beta` erkennt, dass weiteres Erforschen eines Astes keinen Nutzen bringt, wird der Algorithmus den restlichen Ast nicht bewerten und somit Zeit sparen. Gespeicherte `alpha` und `beta` Werte werden jedem Aufruf der Funktion mit soeben ausgerechneten Werten verglichen. Die Alpha-Beta-Suche macht sich die Regel des Minimax-Algorithmus, dass immer der best-gewertete Zug gespielt wird, zunutze indem dieser Vergleich auch über mehrere Äste hinweg gemacht werden kann. So erkennt der Algorithmus weiteres Bewerten eines Astes als unnötig an, weil jegliche neuberechnete Werte `alpha` und `beta` unter keinen Umständen übersteigen werden.

Der um die Alpha-Beta-Suche erweiterte Minimax-Algorithmus sieht als Code-Fragment wie folgt aus:

```
def minimax(depth, alpha, beta, maximizing_player):
    if depth == 0 or game_over():
        evaluate_position(color)
        return evaluation

    if maximizing_player:
        max_eval = -math.inf
        for move in valid_moves:

            move(piece, move)
            evaluation = minimax(depth - 1, alpha, beta, False)
            unmake_move(piece, move)

            max_eval = max(max_eval, evaluation[0])
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = math.inf
        for move in valid_moves:

            minimax_move(piece, move)
            evaluation = minimax_ascended(depth - 1, alpha, beta, True)
            unmake_move(piece, move)

            min_eval = min(min_eval, evaluation[0])
            beta = min(beta, min_eval)
            if beta <= alpha:
                break
        return min_eval
```

Auch wenn die Alpha-Beta-Suche die Laufzeit meines Algorithmus erheblich gesenkt hat, war ich mir von Anfang an der ineffizienten Funktion für das Generieren der validen Spielzüge bewusst. Der Suchalgorithmus durchläuft alle möglichen Spielstellungen im Voraus und muss dementsprechend pro Zug, der gemacht wird, überprüfen, ob das Spiel zu Ende ist und alle validen Züge der neuen Stellung aufs Neue berechnen. Wenn der Zuggenerator und die Berechnung für das Spielende bereits ineffizient sind, werden auch keine Kürzungsversuche des Minimax-Algorithmus, wie mit Hilfe der Alpha-Beta-Suche, die Laufzeiteffizienz signifikant verbessern. Deswegen habe ich mich nach Vollendung des Suchalgorithmus nochmals mit meinem Zuggenerator und anderen Zeit einnehmenden Funktionen befasst, anstatt meine Bewertungsfunktion auszubauen. Denn lieber spiele ich gegen eine schlechtere Engine, die wenigstens in überschaubarer Zeit einen Zug spielt, als gegen eine bessere Engine, die erheblich viel mehr Zeit braucht, um einen Zug zu spielen.

Kapitel 4

Schlussbemerkungen

Rückblickend muss ich gestehen, war diese Arbeit ein sehr langer Weg. Vor einem Jahr fing ich an, mich detaillierter mit der Computer-Programmierung auseinanderzusetzen. Als ein „übermütiges Ich“ sich eine Schach-Engine mit Python zu programmieren zum Ziel gesetzt hatte, war mir das Ausmass der Arbeit überhaupt nicht bewusst. Ich habe mich mit meiner Arbeit ins kalte Wasser gestürzt und habe ohne wirkliches Vorwissen einfach begonnen, zu programmieren. Mit jeder neuen Zeile Code schien ich das Thema besser zu verstehen und das Arbeiten an dem Schachprogramm leichter zu fallen.

Bei einer Möglichkeit die gleiche Arbeit nochmal zu machen, würde ich unbedingt einerseits die Programmiersprache zu C++ ändern und andererseits meine Schachbrett-Darstellung mit Bitboards umsetzen. Mit C++ wäre die Programmierung einer schnelleren Schach-Engine möglich gewesen, was einerseits an der Effizienz der Programmiersprache liegt und andererseits an der Menge von Informationen, die auf dem Internet erhältlich sind. Die Anwendung von Bitboards hätte die Berechnung einiger zeitintensiven Funktionen erheblich beschleunigen können. Dürfte ich nur einen Erfahrungswert meiner Arbeit mitnehmen, wäre es das Debuggen. Mir war am Anfang der Arbeit nicht bewusst, wie viel Zeit ich aufwenden würde, um einen Bug im Programm ausfindig zu machen und dann noch zu beseitigen. Um ehrlich zu sein, habe ich einen grossen Teil meiner Arbeit damit verbracht, Bugs zu beheben. Dementsprechend habe ich Folgendes aus Sicht der Programmierung gelernt:

1. Sich im Vorhinein klarmachen, was für einen Code man implementieren möchte, und sich schon mögliche kritische Aspekte des Codes im Kopf ausmalen.
2. Im Falle eines Bugs schnell den Ort des Fehlers ausmachen können; einerseits durch die Fehlermeldungen im Code und andererseits mit Hilfe von Print-Statements oder Debugging-Tools.

Allgemein möchte ich festhalten, dass die Programmierung eines umfassenden Schachprogramms im Rahmen einer Maturaarbeit aufgrund des zeitlichen Rahmens nur eingeschränkt möglich ist, zumindest mit bedingten Programmierkenntnissen. Daher gibt es auch einiges an Verbesserungspotenzial für mein entwickeltes Schachprogramm. Zum Beispiel kann die Schach-Engine ausgebaut werden; entweder durch die Verwendung anderer Programmiersprachen oder durch eigene Implementierungen, die auf spezielle Datenstrukturen wie den oben erwähnten Bitboards basieren.

Die graphische Darstellung des Schachbrettes und der Schachfiguren bietet natürlich auch Optionen zur optischen Aufwertung. Zum Beispiel kann mit attraktiveren Farben

oder Schriftzügen gearbeitet werden oder eine Spieluhr hinzugefügt werden, die den Spieler davon abhält, sich zu lange, zu viele Gedanken über einen Zug zu machen. Darauf aufbauend könnten dann Zeitlimits für Spieler konfiguriert und im Spiel durch Erfassen der jeweiligen Zeiten erzwungen werden.

Als Programmieranfänger habe ich natürlich alle typischen Anfängerfehler gemacht. Nach meinen Erfahrungen würde ich in Zukunft mehr darauf achten, das Programm mit Hilfe von möglichst kleinen Funktionen (Bausteinen) strukturierter umzusetzen. Eine verbesserte Strukturierung erleichtert einerseits das Verständnis für den Code, andererseits hilft es ungemein bei der Fehlersuche und dem Debugging. Das Gleiche gilt auch für die schriftliche Zusammenfassung der Arbeit. Mit einer frühzeitigeren Strukturierung der Abschnitte hätte ich mir wohl einiges an Zeit sparen können. Diese schriftliche Arbeit dokumentiert nur einen Teil des tatsächlich umgesetzten Programms.

Mit der Fertigstellung dieser Maturaarbeit habe ich die zentralen Ziele, die ich mir anfangs vorgenommen hatte, vollständig erreicht. Ich habe mich in ein für mich neues Gebiet komplett selbständig eingearbeitet und dabei einiges über das Entwickeln von Computer-Programmen und über die Programmiersprache Python gelernt. Ich konnte mich in die Funktionweise von Game-Entwicklungs-Tools wie Pygame vertiefen, also das Frontend von Spielprogrammen nachvollziehen und habe mich gründlich mit dem Backend, dem Kern meiner Arbeit, auseinandergesetzt. Ausserdem habe mich das erste Mal tiefergehend mit dem Textverarbeitungsprogramm L^AT_EX befasst, welches ich sicherlich als eine attraktive Option für zukünftige Arbeiten in Betracht ziehen werde.

Würde ich jedoch eine Kosten-Nutzen-Rechnung aufstellen, die den Faktor Zeit angemessen berücksichtigt, müsste ich nachfolgenden Kolleginnen und Kollegen eher von einer solchen Programmierarbeit abraten, es sei denn, man verfügt bereits über vertiefte Programmierkenntnisse und hat bereits Erfahrungen im Bereich Schachprogrammieren gemacht. So war das Erstellen eines Schachprogramms mit Python zwar sehr spannend und lehrreich, aber mit einem enormen Aufwand verbunden.

Danksagungen: Abschliessend möchte ich mich bei allen bedanken, die zu dieser Arbeit in irgendeiner Art beigetragen haben. Mein Betreuer Severin Walser inspirierte mich zu dieser Arbeit und unterstützte mich sehr mit seinem Feedback. Vielen Dank dafür. Auch danken möchte ich Stefan Müller, der sich als Korreferent zur Verfügung gestellt hat. Ein Dankeschön geht an meine kleine Schwester Tara, die in einem Testspiel meine KI vernichtend geschlagen hat und mir mit ihren Testspielen half, versteckte Bugs zu finden. Zuletzt möchte ich mich noch bei meinen Eltern für das finale Korrekturlesen bedanken.

Literaturverzeichnis

- [1] Schachtürke, <https://de.wikipedia.org/wiki/Schachtürke>.
- [2] Türken (Verb), [https://de.wikipedia.org/wiki/Türken_\(Verb\)](https://de.wikipedia.org/wiki/Türken_(Verb))/.
- [3] In 1950, Alan Turing Created a Chess Computer Program That Prefigured A.I., <https://www.history.com/news/in-1950-alan-turing-created-a-chess-computer-program-that-prefigured-a-i>.
- [4] Schachcomputer, <https://de.wikipedia.org/wiki/Schachcomputer/>.
- [5] DeepBlue, Schachcomputer von IBM, https://de.wikipedia.org/wiki/Deep_Blue/.
- [6] AlphaZero and MuZero, <https://deepmind.google/technologies/alphazero-and-muzero/>.
- [7] Computerschach, <https://de.wikipedia.org/wiki/Computerschach>.
- [8] Python, <https://www.python.org/>.
- [9] pygame documentation, <https://www.pygame.org/docs/ref/pygame.html>.
- [10] How Chess Games Can End: 8 Ways Explained, <https://www.chess.com/article/view/how-chess-games-can-end-8-ways-explained>.
- [11] Chess Board Color Palette, <https://www.color-hex.com/color-palette/8548>.
- [12] Coding Adventure: Chess, <https://www.youtube.com/watch?v=U4ogK0MIzqk>.
- [13] Building my own Chess Engine, <https://healeycodes.com/building-my-own-chess-engine/>.
- [14] Algorithms Explained – minimax and alpha-beta pruning, <https://www.youtube.com/watch?v=l-hh51ncgDI>.