Thema Studienarbeit

Entwicklung einer KI für das Schach-Endspiel

im Studiengang

Angewandte Informatik

an der *Dualen Hochschule Baden-Württemberg Mannheim*

von

Namen: Lukas Becker; Nico Miller

Abgabedatum: 14.04.2022

Bearbeitungszeitraum: 18.10.2021 – 19.04.2022

Matrikelnummern, Kurs: 4836372, 5487547, TINF19AI1

Ausbildungsfirma: Roche Diagnostics GmbH

Betreuer der Hochschule: Prof. Dr. Karl Stroetmann

Ehrenwörtliche Erklärung

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

”Entwicklung einer KI für das Schach-Endspiel"

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

Ort, Datum	*Unterschriften*

Abbildungsverzeichnis

Anmerkung

Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit für alle personenbezogenen Begriffe (z.B. Nutzer etc.) nur die männliche Sprachform verwendet. Sämtliche Personenbezeichnungen gelten gleichermaßen für jedes Geschlecht.

Inhaltsverzeichnis

1 Einleitung

Bevor in die Umsetzung der Aufgabe eingestiegen wird, werden einige Grundlagen des Schachspiels für den ungeschulten Spieler erklärt. Dadurch können die Schritte zur Berechnung eines optimalen Endspiels besser nachvollzogen werden.

2 Schach

Bei Schach handelt es sich um ein Brettspiel, das insgesamt von zwei Spielern gespielt werden kann. Im folgenden Abschnitt soll eine Einführung in den Spielablauf gegeben werden. Diese wird benötigt, um die im Verlauf dieses Dokuments vorgestellten Konzepte zu verstehen.

2.1 Spielbrett und Spielfiguren

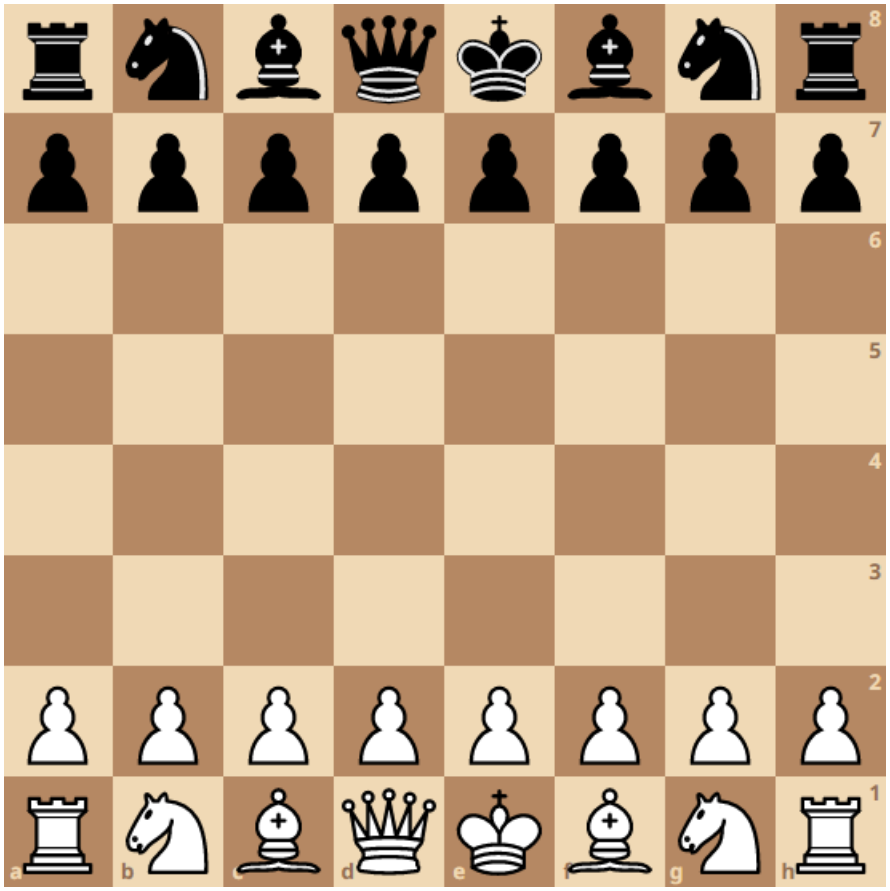
Ein Schachbrett besteht aus insgesamt 64 Feldern, die in einer 8x8 Matrix angeordnet sind. Dabei werden die Spalten mit den Buchstaben a-h und die Zeilen mit den Zahlen 1-8 beschriftet. Das typische Schachmuster entsteht durch einen regelmäßigen Wechsel zwischen weißen und schwarzen bzw. dunklen und hellen Feldern. Dementsprechend sind insgesamt 32 weiße und 32 schwarze Felder auf dem Schachbrett zu finden. Alle Felder können mithilfe der zuvor genannten Beschriftung eindeutig identifiziert werden. Wenn in diesem Dokument Felder durch ein Kürzel wie beispielsweise “c5” spezifiziert werden, ist das Feld gemeint, das in der Spalte “*Buchstabe*” und der Zeile “*Zahl*” zu finden ist. Das vorherige Beispiel befindet sich also in der dritten (c) Spalte und der fünften (5) Zeile.

Im Folgenden werden die Spalten des Schachbretts als Linien und die Zeilen als Reihen bezeichnet.

Dieses Schachbrett wird maximal von 32 Figuren besetzt (16 weiße und 16 schwarze Figuren). Eine Seite besteht aus insgesamt sechs unterschiedlichen Figuren. Bei diesen Figuren handelt es sich um:

- Bauern (8)
- Springer (2)
- Läufer (2)
- Türme (2)
- Dame (1)
- König (1)

Die Zahl in den Klammern steht hierbei für die Anzahl an Figuren pro Spieler. Die Aufstellung zu Beginn des Spiels kann aus der folgenden Abbildung entnommen werden:

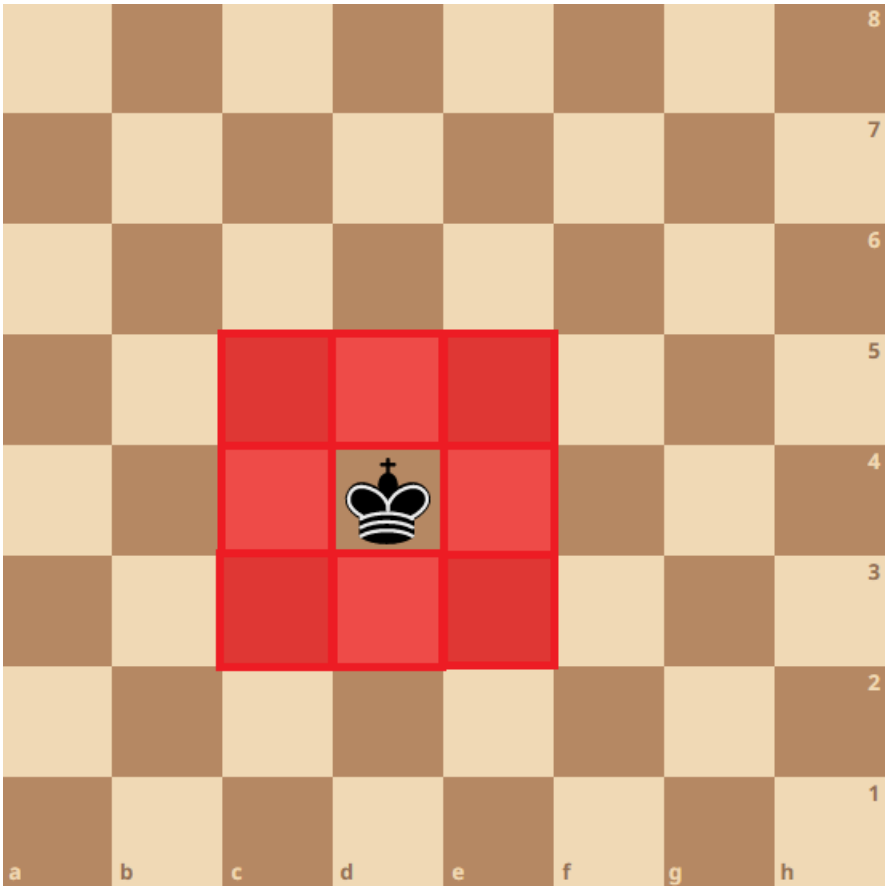


2.1.1 Bewegungsmöglichkeiten der Figuren

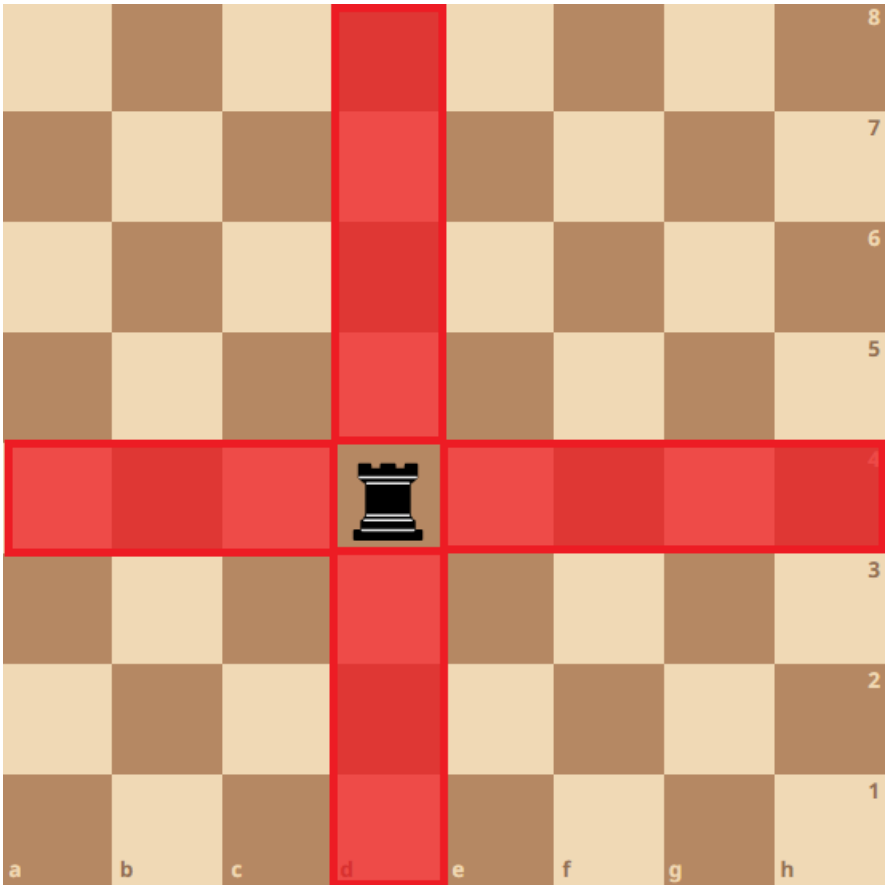
Jede der bereits genannten Figuren hat einen Wert und ein Bewegungsmuster. Figuren können im Rahmen dieses Bewegungsmuster bewegt werden, werden aber durch andere Figuren blockiert. Eine Figur kann nur in Ausnahmefällen übersprungen werden und blockiert grundsätzlich die Bewegungen aller anderen Figuren. Figuren des Gegners (der anderen Farbe) können geschlagen werden, indem eine eigene Figur auf dasselbe Feld gestellt wird. Eine geschlagene Figur wird vom Spielfeld entfernt. Zwei Figuren derselben Farbe können nicht auf demselben Feld platziert werden. Diese lauten wie folgt (Jonathan Carlstedt, Die kleine Schachschule (2015): S.10ff., S.40):

- **König** (unendlich): Der König gehört zu den unbeweglichsten Figuren auf dem Spielfeld. Er kann pro Zug nur ein Feld entlang einer Reihe, Linie oder Diagonalen bewegt werden. Dadurch besitzt er jedoch die Möglichkeit, in jede Richtung eine gegnerische Figur zu schlagen. Als weitere Einschränkung muss beim Ziehen mit dem König beachtet werden, dass das angestrebte Feld nicht durch eine gegnerische Figur abgedeckt wird. Ein Feld gilt als abgedeckt, wenn eine geg-

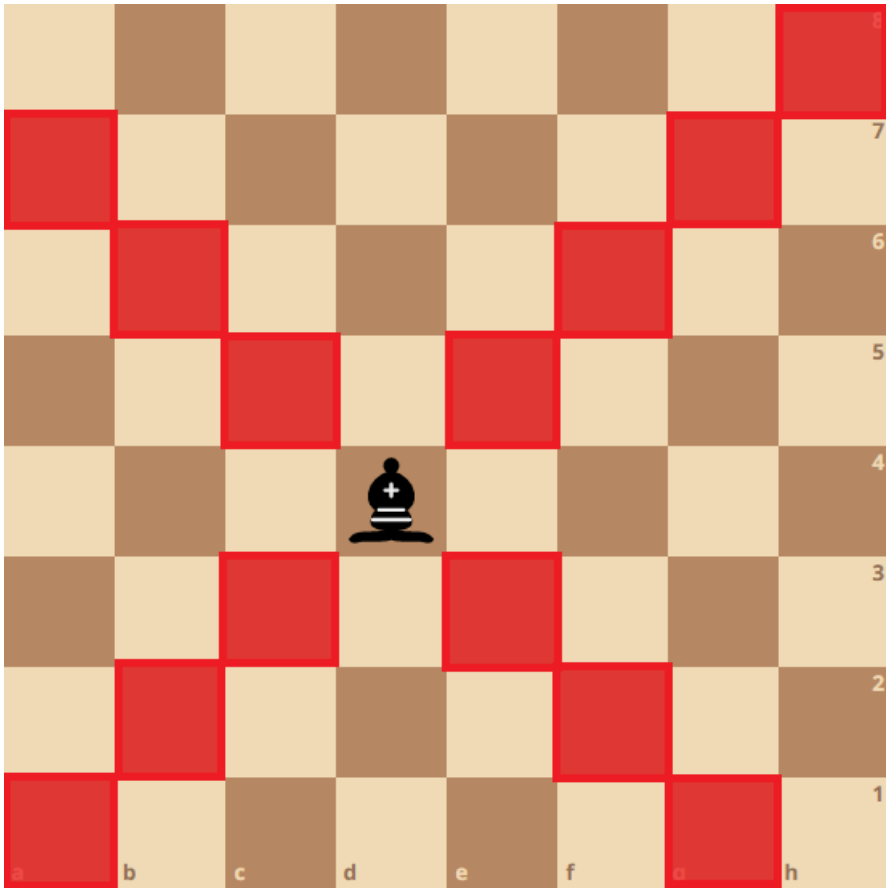
nerische Figur es in einem Zug betreten und die darauf stehende Figur schlagen kann. Ist dies der Fall, darf der König nicht auf dieses Feld gesetzt werden.



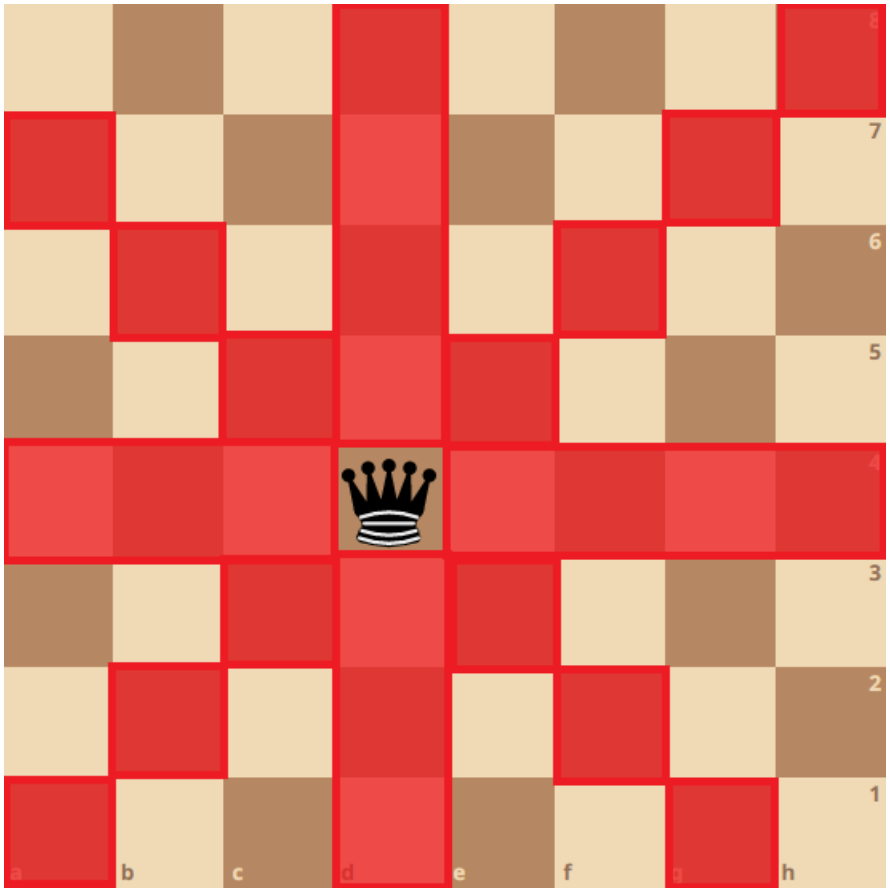
- **Turm** (5): Der Turm besitzt die Möglichkeit, in einer Reihe oder Linie beliebig viele Felder zu überqueren (maximal bis zum Ende des Spielfeldes).



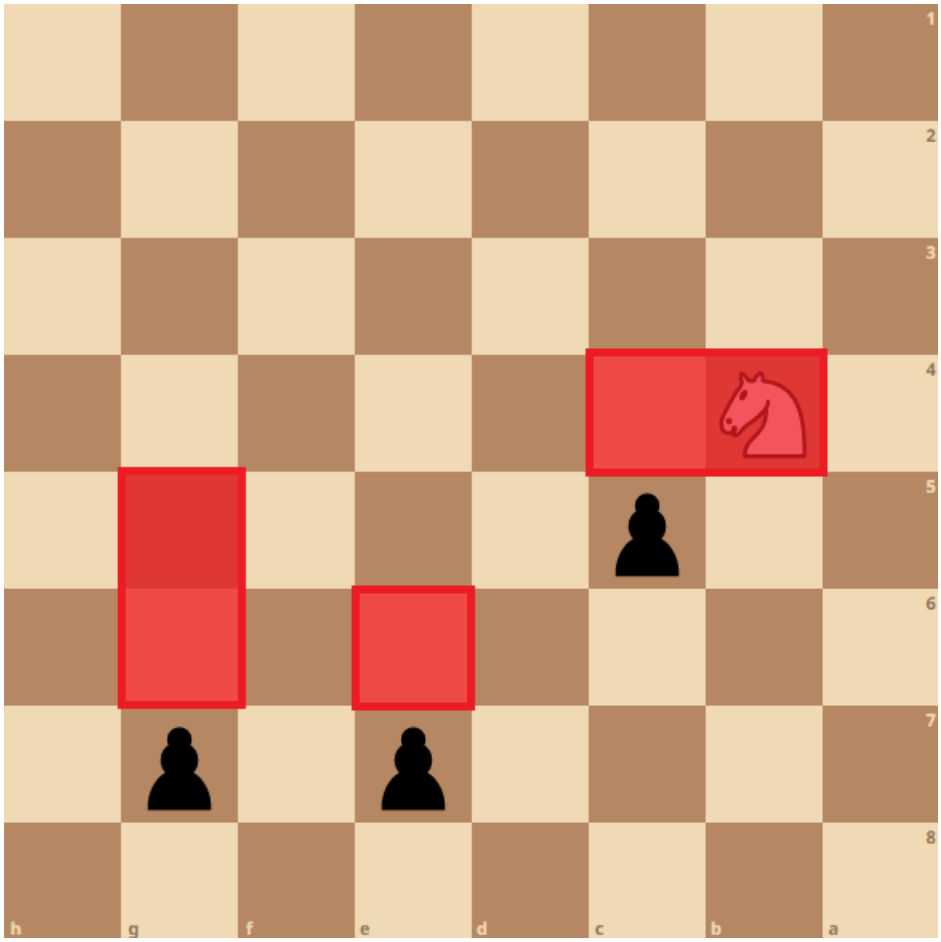
- **Läufer** (3): Der Läufer kann wie ein Turm in geraden Linien bewegt werden. Er unterscheidet sich dadurch, dass er nur diagonal bewegt werden kann. Ein Läufer auf dem Feld “a1” kann folglich nur nach “b2” oder entlang der Diagonale bewegt werden.



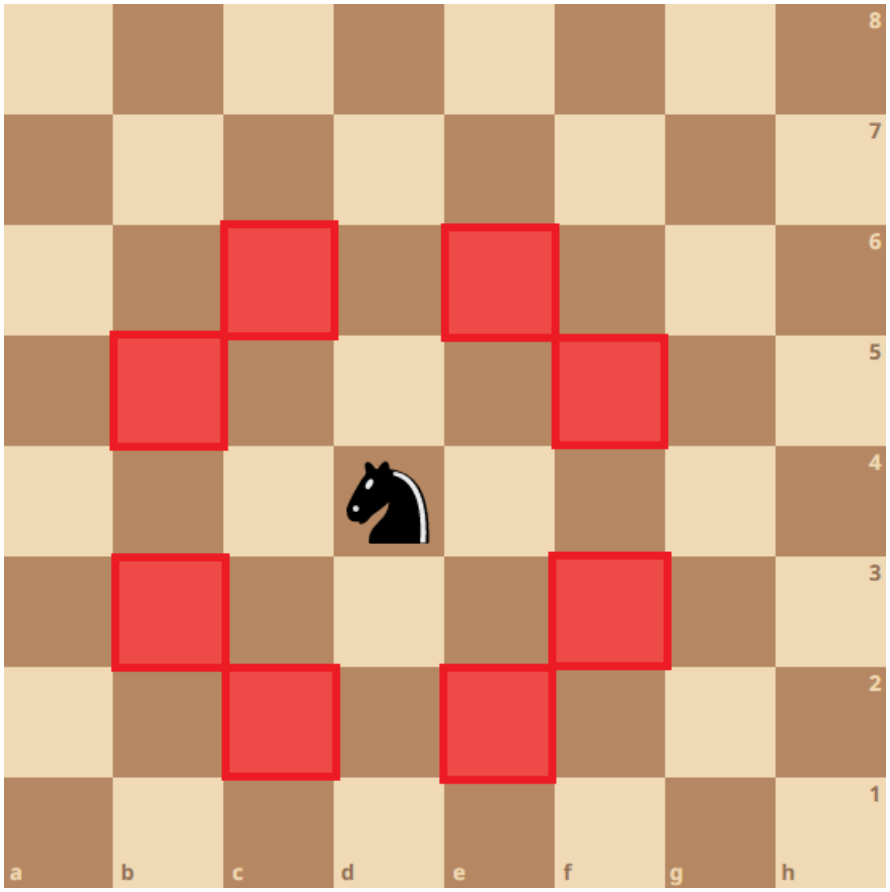
- **Dame** (9): Die Dame zählt zu den beweglichsten Figuren auf dem Spielfeld. Sie kombiniert die Bewegungsmuster des Läufers und des Turms. Das bedeutet, dass sie horizontal (entlang der Reihen), vertikal (entlang der Linien) und diagonal bewegt werden kann.



- **Bauer** (1): Der Bauer ist die Figur mit der geringsten Beweglichkeit. Dieser kann nur entlang der Linie nach vorne bewegt werden. Die erste Bewegung jedes Bauern kann ein oder zwei Felder weit sein, folgende Bewegungen sind immer genau ein Feld weit. Eine Besonderheit des Bauerns, liegt in der Richtung, in die ein Bauer gegnerische Figuren schlagen darf. Dieser darf nur diagonal nach vorne schlagen. Weiter wird der Bauer in eine beliebige Spielfigur (außer einem Bauern und einem zweiten König) gewandelt, sobald er die Grundlinie des Gegners erreicht hat.



- **Springer** (3): Der Springer besitzt im Gegensatz zu allen bereits beschriebenen Figuren keine lineare Bewegungsrichtung. Er kann um zwei Felder nach vorne und ein Feld zur Seite versetzt werden. Dieses Verfahren gilt in jede Richtung, sodass der Springer im Optimalfall acht Felder erreichen kann. Der Name des Springers kommt dadurch zustande, dass er die einzige Figur ist, die andere Figuren überspringen kann. Nur das “Zielfeld” kann durch eine eigene Figur blockiert werden.



2.2 Spielablauf

In einem Spiel ziehen die Spieler immer abwechselnd eine Figur ihrer Farbe. Den ersten Zug hat dabei immer weiß.

Beide Spieler verfolgen während der ganzen Partie das Ziel, den gegnerischen Spieler Schach-Matt zu setzen. Ein Spieler ist Schach-Matt, wenn folgende Bedingungen erfüllt sind: 1. Der König wird durch eine gegnerische Figur bedroht. 2. Der König kann dieser Bedrohung nicht ausweichen.

Eine solche Bedrohung liegt vor, wenn der gegnerische Spieler im nächsten Zug den König schlagen kann. Dies kann auf drei unterschiedliche Weisen pariert werden: 1. Der König bewegt sich aus dem “Schach”. 2. Der Spieler schlägt die Schach-gebende Figur. 3. Eine Figur stellt sich zwischen die Schach-gebende Figur und den König.

Ein anderer Spielausgang neben dem Schach-Matt liegt in dem Patt. Ein Patt ist dann gegeben, wenn der Spieler, der am Zug ist, keine Figur mehr ziehen kann und der König des Ziehenden nicht im Schach steht.

Da diese Studienarbeit nicht vorsieht, das komplette Schachspiel zu erklären, werden die restlichen Spielregeln nicht näher erläutert. Diese werden aber in der python-chess

Bibliothek, welche für die Abbildung des Schachspiels im Code verwendet wird, umgesetzt und berücksichtigt.

Anhand der Nummerierung der Notebooks können nun die einzelnen Schritte zur Erstellung und Validierung einer Schach-Endspiel-KI nachvollzogen werden. Die einzelnen Notebooks begleiten chronologisch die Erstellung, Validierung und Verwendung einer Endspiel-KI. Die hierfür notwendigen Notebooks sind:

1. 02_calculation.ipynb
2. 03_play_against_ai.ipynb
3. 04_play_from_history.ipynb
4. 05_stockfish_compare.ipynb
5. 06_validate_sequences.ipynb

Im Util-Ordner werden zusätzlich allgemeine Importe und Funktionen aufgelistet, die für die Ausführung der aufgelisteten Notebooks benötigt werden.

In der PDF-Version dieser Arbeit ist jedes der aufgelisteten Notebooks ein Kapitel.

3 Berechnung der Endspieldatenbank

Wie in Notebook `01_chess_introduction` bereits erklärt, ist ein Schachspiel gewonnen, wenn die gegnerische Figur mattgesetzt wurde.

Bei einer geringen Anzahl an Figuren P im Spielzustand lassen sich alle möglichen Positionen berechnen. Aus diesen kann eine Strategie entwickelt werden, den Gegner zu schlagen.

Im weiteren Verlauf werden folgende Definitionen verwendet:

- $board.pieces$: Liste der Figuren, welche in einem Zustand vorhanden sind.
- $valid\_boards$: Alle Zustände des Schachspiels, die gegen keine Regeln verstoßen.
- won_boards: Alle Zustände des Schachspiels, in denen ein Spieler gewonnen hat.
- $previous\_states(b)$: Alle Zustände, aus denen durch Ausführen eines einzelnen Zuges der Zustand b erreicht werden kann.

Seien alle möglichen (validen) Kombinationen von Positionen der Figuren P die Menge S.

Für S gilt:

$$board \in S \implies \forall p \in P : p \in board.pieces \land board \in S \implies board \in valid\_Boards$$

Aus der Menge S lassen sich Zustände auswählen, welche n Züge vom Sieg entfernt sind. Diese Zustände lassen sich in S_n zusammenfassen. Ist das Spiel gewonnen, verbleiben 0 Züge bis zum Sieg.
Für alle diese Zustände, in denen ein Spieler mattgesetzt ist, gilt:

$$board \in S_0 \implies board \in won\_boards \land board \in S$$

Aus dieser Definition können induktiv die verbleibenden S_n hergeleitet werden:

$$board \in S_{n+1} \iff board \in S \land \exists b \in S_n : board \in previous\_states(b)$$

Für die Berechnungen in diesem Notebook gilt da für den schwarzen Spieler immer nur der König auf dem Feld steht weiter Folgendes:
$board.turn$: Der Spieler, welcher am Zug ist.

$$n \equiv 0 \mod 2 \implies \forall b \in S_n : b.turn = schwarz$$

$$\land$$

$$n \equiv 1 \mod 2 \implies \forall b \in S_n : b.turn = weiss$$

Dieses Notebook wird zur Berechnung der S_n Mengen verwendet. Diese werden benötigt, um letztendlich ein Schach-Endspiel lösen zu können.

3.1 Ein Hinweis zu Spiegelungen

In diesem Notebook werden Spiegelungen der Situationen verwendet. Die technische Umsetzung dieser Spiegelungen werden im Verlauf des Dokuments erklärt, an dieser Stelle soll lediglich eine Einführung in die Theorie hinter dem Spiegeln von Situationen erklärt werden.

Durch die zuvor erklärten Bewegungsmuster der Figuren sind Schachbretter in vielen Fällen symmetrisch.

Eine Position mit dem Turm in “a8”, der Dame in “g6” und dem gegnerischen König in “h8” ist genauso verloren wie dieselbe Position nur mit dem Turm in “a1”, der Dame in “g3” und dem König in “h1”. Dies wäre eine Spiegelung entlang der horizontalen

zwischen den Zeilen 4 und 5. Weiter sind auch Spiegelungen entlang der vertikalen (Zwischen Reihe e und f), den Diagonalen und Rotationen (jeweils um 90°, 180° und 270°) möglich.

Durch das simple Spiegeln der Spielsituationen können aus einer validen Spielsituation bis zu sieben weitere ohne großen Rechenaufwand erstellen. Aus diesem Grund werden in diesem Dokument bei jeder Berechnung neuer Situationen diese gespiegelt und die Spiegelungen ebenfalls überprüft und abgespeichert.

Da Bauern sich nur in eine Richtung bewegen können, werden nur Spielsituationen gespiegelt, welche keine Bauern enthalten.

3.2 Ein Hinweis zur effizienten Ergebnisverwaltung

Im Verlauf der Berechnung muss mehrfach überprüft werden, ob eine Situation bereits bekannt und einem S_n zugeordnet ist. Da der Abgleich mit einer Liste in Python ineffizient ist, findet dieser Abgleich mit Mengen statt. Mengen werden in Python als Hash-Tabellen umgesetzt und haben damit eine Zeitkomplexität bei der Überprüfung, ob sie ein bestimmtes Element enthalten von $\mathcal{O}(1)$. `board` Objekte der `chess` Library sind jedoch nicht "Hashbar". Im Sinne der in dieser Arbeit getätigten Berechnungen reichen die Informationen über die Stellung der Figuren und dem Spieler, welcher am Zug ist, aus. Es wird daher für die Verwendung in Python Mengen mit einer Tupel-Repräsentation der Situationen wie folgt gearbeitet:

$$Tupel := < board.turn, board.\_\_str\_\_() >$$

Die Funktion `board.__str__()` gibt einen String zurück, welcher ein Schachbrett wie folgt darstellt:

```
'r n b q k b n r\np p p p p p p p\n. . . . . .\n. . . . . .\n. . . . . .\
```

Die Eigenschaft `board.turn` ist ein boolscher Wert. Sowohl Tupel, als auch Strings und boolsche Werte sind in Python "Hashable", weshalb diese Darstellung in Mengen verwendet werden kann.

Um die Effizienz weiter zu steigern, berechnet dieses Notebook nicht alle Situationen S und entfernt daraus die Situationen für ein S_n wie in der Aufgabenstellung beschrieben. Stattdessen werden alle bekannten Situationen in `used_boards` gespeichert. Dopplungen werden also nicht vermieden indem Situationen aus einer großen Liste entfernt werden, sondern eine Liste der entfernten Situationen geführt und neue Situationen mit dieser abgeglichen.

3.3 Funktionen zur Bestimmung aller gültigen Positionen

Wie bereits in `01_chess_introduction` beschrieben, besteht ein Schachbrett aus insgesamt acht Spalten und Zeilen. Die Spalten werden durch Buchstaben gekennzeichnet, die Zeilen durch Zahlen. Aus der Kombination einer Spalte (z.B. a) und einer Zahl (z.B. 1) erhält man eine eindeutige Kennzeichnung für ein Feld (z.B. a1).
Die folgende Funktion kombiniert die Buchstaben a bis h mit den Zahlen 1 bis 8 zu Feldnamen und gibt diese zurück.

[ ]:
```
def get_all_squares():
    columns = {
        1 : 'a',
        2 : 'b',
        3 : 'c',
        4 : 'd',
        5 : 'e',
        6 : 'f',
        7 : 'g',
        8 : 'h'
    }

    all_squares = []
    for row in range(1,9):
        for col_num in range(1,9):
            column = columns[col_num]
            all_squares.append(chess.parse_square(column + str(row)))

    return all_squares
```

Das Erstellen jeglicher Boards wird mit der Funktion `place_piece_everywhere_on_every_board` umgesetzt. Diese erhält folgende Parameter:

- `piece`: Die zu platzierende Figur als Objekt der chess-Library.
- `list_of_boards`: Eine Liste mit Board-Objekten, auf welchen die Figur platziert werden soll.

Die Funktion betrachtet jede Situation in der `list_of_boards`. Das übergebene `piece` wird auf jeden freien Platz dieser Situation platziert. Jedes Mal, wenn eine Figur platziert wird, wird eine Kopie des Board-Objektes erstellt, die `list_of_boards` wird folglich nicht verändert.

Wenn der zweite König platziert wird, wird die Situation zusätzlich auf Validität überprüft. Wenn alle Figuren platziert wurden, werden nur Boards, in denen Schwarz matt ist, zurückgegeben.

Um die Effizienz der Berechnung zu erhöhen, werden die Schachbretter gespiegelt. Damit bei folgenden Berechnungen Situationen nicht einmal durch Spiegelung und einmal durch Bewegung von Figuren erreicht werden, werden die ungespiegelten Situationen in der Liste `uniques` gespeichert.

Die Funktion gibt als Ergebnis eine Liste aller generierten Zustände als `result_list`, die Menge der bereits verwendeten Situationen `used_boards` und alle ungespiegelten Boards (vor Spiegelungen) `uniques` zurück.

[ ]:
```
def place_figure_everywhere_on_every_board(piece, list_of_boards,␣
↪piece_count, user_wants_pawn):
    result_list = []
    uniques = []
    all_squares = get_all_squares()
    used_boards = set()

    for board in list_of_boards:
        squares_used = list(board.piece_map().keys())
        for square in all_squares:
            if square not in squares_used:
                tmp_board = board.copy()
                tmp_board.set_piece_at(square, piece)

                if len(squares_used) > 1 and not tmp_board.is_valid():
                    # Don't process invalid boards further
                    # than the second king
                    continue

                outcome = tmp_board.outcome()
                if outcome is not None:
                    rep = (tmp_board.turn, tmp_board.__str__())
                    if outcome.winner is not None and tmp_board.
↪is_valid() and rep not in used_boards:
                        uniques.append(tmp_board)
                        result_list.append(tmp_board)
```

```
                        used_boards.add((tmp_board.turn,tmp_board.
↪__str__()))
                        if not user_wants_pawn:
                            for swt in Swap_Type:
                                mir_board = mirror(tmp_board, swt)
                                if (mir_board.turn, mir_board.
↪__str__()) not in used_boards:
                                    result_list.append(mir_board)
                                    used_boards.add((mir_board.
↪turn,mir_board.__str__()))
                        continue

                if len(squares_used) + 1 < piece_count: #Board is␣
↪valid, but needs more pieces
                    result_list.append(tmp_board)
    return result_list, used_boards, uniques
```

3.4 Die Ursprungsmenge S_0 erstellen

Als Basis der Berechnung dient die Liste S_0. Diese enthält alle möglichen Konstellationen der Spielfiguren auf dem Spielbrett, in denen Weiß Schwarz besiegt hat. Hierfür werden die Figuren mit der Funktion `place_figure_everywhere_on_every_board` auf allen Positionen platziert.

Die Funktion `setup_boards` automatisiert dies und gibt die Liste S_0, eine Menge der bereits bekannten Situationen `used_boards` und die Menge der ungespiegelten Situationen `uniques` zurück.

[ ]:
```
def setup_boards(user_supplied_pieces, user_wants_pawn):
    pieces_to_place = create_piece_list(user_supplied_pieces)

    empty_board = chess.Board().empty()
    empty_board.turn = chess.BLACK
    s_0 = [empty_board]

    piece_count = len(pieces_to_place)
    for piece in pieces_to_place:
```

```
        s_0, used_boards, uniques =␣
↪place_figure_everywhere_on_every_board(piece, s_0, piece_count,␣
↪user_wants_pawn)

    print(str(len(s_0)) + " Boards in S_0")
    return s_0, used_boards, uniques
```

Bevor die Boards für S_0 erstellt werden können, müssen die vom Nutzer getätigten Eingaben zu den zu verwendeten Figuren mit den immer vorhandenen Figuren kombiniert werden. Weiter wird ein eingegebener Bauer durch eine Königin ersetzt. Ein Endspiel mit zwei Königen und einem Bauer kann nicht gewonnen werden, weshalb keine Situationen für S_0 gefunden werden würden. Die Königin wird später im Ablauf wieder durch einen Bauern ersetzt. Die Theorie hinter diesem Tausch wird zu einem späterem Zeitpunkt erklärt.

[ ]:
```
# A queen will automatically be replaced by a pawn
def create_piece_list(user_supplied_pieces):
    if chess.Piece.from_symbol("P") in user_supplied_pieces:
        user_supplied_pieces.remove(chess.Piece.from_symbol("P"))
        user_supplied_pieces.append(chess.Piece.from_symbol("Q"))

    return [chess.Piece.from_symbol("K"), chess.Piece.from_symbol("k")]␣
↪\
           + user_supplied_pieces
```

3.5 Rückwärts neue Situationen bestimmen

Der nächste Schritt besteht darin, sämtliche S_n Mengen zu bestimmen. Hierzu wird eine bereits bestimmte S_n Menge genommen und alle Situationen berechnet, die durch Durchführen eines Zugs zu einer Situation aus S_n werden.

Besonders muss bei dieser Art der Bestimmung auf die Einordnung der Situationen, bei welchen Schwarz am Zug ist, geachtet werden. Da beim späteren Verwenden der KI die Züge des schwarzen Spielers nicht beeinflusst werden können, muss jeder mögliche Zug einer Situation in S_{n+1} mit $n\%2 = 0$ zu einer Situation aus S_n führen. Diese Überprüfung wird mit der Funktion `check_black_determinism` durchgeführt. Da die Züge von Weiß gezielt gewählt werden können, ist diese Überprüfung bei $n\%2 = 1$ nicht nötig.

Außerdem müssen bei der Durchführung des Algorithmus weitere Aspekte berück-

sichtigt werden:

- Da Bauern nur in eine Richtung laufen können, müssen die rückwärts Schritte eines Bauern manuell durchgeführt werden. Bauern werden daher im ersten Schritt ignoriert.
- Bauern, die die oberste Reihe des Spielfeldes erreichen, können zu einer anderen Figur eingetauscht werden. Dieser Schritt wird nicht durch die Pseudo-Legal-Moves abgedeckt, daher wird, sollte sich eine Königin in der obersten Reihe befinden, diese manuell durch einen Bauern ersetzt.

Die Umsetzung erfolgt durch die Funktion `previous_states`. Alle Funktionsparameter können aus der nachfolgenden Liste entnommen werden:

- `used_boards`: Die Menge aller bereits einem n zugeordneten Situationen, welche nicht noch einmal beachtet werden sollen.
- `iteration_count`: n des S_n, welches gerade berechnet wird.
- `user_wants_pawn`: Ein Flag, welches steuert, ob spezifische Bewegungen des Bauern berechnet werden sollen.
- `uniques`: Die Situationen, welche als Ursprung der Spiegelung verwendet werden.

Der Algorithmus zur Bestimmung der Menge S_{n+1} wird im folgenden Abschnitt beschrieben. Um die Funktion übersichtlicher zu halten, wurden teile des Algorithmus in die Funktion `moves` übertragen.

- Über die Uniques (ungespiegelte Situationen) iterieren.
  - Den Spieler, welcher am Zug ist, wechseln (Da, um im aktuellen Zustand anzukommen, der andere Spieler einen Zug gemacht hat)
  - Alle Positionen mit Bauern berechnen
  - Alle pseudo-legalen Bewegungen mittels der Funktion `regular_moves` durchführen. Hierbei werden keine Züge der Bauern beachtet. Die technische Umsetzung wird in der Dokumentation der Funktion erklärt.
  - Wenn die Bewegungen von Bauern abgebildet werden müssen:
    - Bauern manuell einen Schritt “nach hinten” setzen.
    - Überprüfen, ob eine Dame in der obersten Reihe durch einen Bauern in der vorletzten ersetzt werden muss.
    - Die technische Umsetzung dieser Aktionen wird in der Dokumentation der Funktionen `pawn_moves` und `replace_queen_with_pawn` erklärt.
  - Den Spieler, welcher ursprünglich am Zug war, wiederherstellen.
  - Wenn $(n+1)\%2 = 0$ überprüfen, ob alle zuvor berechneten Boards mit allen Moves in S_n enden.

Außerdem müssen bei der Durchführung des Algorithmus weitere Aspekte berücksichtigt werden:

- Da Bauern nur in eine Richtung laufen können, müssen die rückwärts Schritte eines Bauern manuell durchgeführt werden. Bauern werden daher im ersten Schritt ignoriert.
- Bauern, die die oberste Reihe des Spielfeldes erreichen, können zu einer anderen Figur eingetauscht werden. Dieser Schritt wird nicht durch die Pseudo-Legal-Moves abgedeckt, daher wird, sollte sich eine Königin in der obersten Reihe befinden, diese manuell durch einen Bauern ersetzt.

Die Funktion bestimmt die Menge S_{n+1}, die Menge der bekannten Boards als Tupel `used_boards` und die ungespiegelten Origniale aus $n+1$ `s_n1_uniques`

[ ]:
```
def previous_states(used_boards, iteration_count, user_wants_pawn,␣
↪uniques):
    #variables
    s_n1 = []
    s_n1_tuples = set()
    s_n1_uniques = []
    s_n1_uniques_tuples = set()

    for i in range(len(uniques)):
        status = "Calculating S" + str(iteration_count) + " - Board " +␣
↪str(i+1) + " of " + str(len(uniques)) + " from S" +␣
↪str(iteration_count-1)
        clear_output(wait=True)
        print(status)

        # Copy current board and invert the player
        chess_board = uniques[i].copy()
        chess_board.turn = chess_board.turn ^ True

        # Find all Pawns
        pawn_positions = find_pawns(chess_board)

        # try moves and check if they lead to new boards
        s_n1, s_n1_tuples, s_n1_uniques, s_n1_uniques_tuples =␣
↪moves(chess_board, used_boards, s_n1_tuples, pawn_positions,␣
↪user_wants_pawn, s_n1, s_n1_uniques, s_n1_uniques_tuples)
```

```
            # Restore the original state of the board
            chess_board.turn = chess_board.turn ^ True

        # Only needed for Black-Moves
        if iteration_count % 2 == 0:
            clear_output(wait=True)
            print("Calculating S" + str(iteration_count) + " - Checking␣
↪Black Moves for determinism")

            s_n1, s_n1_tuples, s_n1_uniques = check_black_determinism(s_n1,␣
↪used_boards, s_n1_uniques_tuples)

        clear_output(wait=True)
        print("Done with S" + str(iteration_count))
        return s_n1, used_boards | s_n1_tuples, s_n1_uniques
```

3.6 Hilfsfunktionen für die Berechnung

Die folgenden Funktionen werden zur Berechnung der previous_states verwendet. Sie übernehmen dabei diverse Aufgaben wie das Durchführen von regulären Moves oder das “manuelle” Versetzen von Figuren, um eine andere Situation zu generieren.

Die Funktion `moves` führt für eine Situation `chess_board` zunächst Züge mit allen Figuren außer dem Bauern durch. Wenn der Nutzer einen Bauern in seiner Konfiguration angegeben hat, werden auch Bauernzüge sowie der Tausch Dame zu Bauer durchgeführt.

[ ]:
```
def moves(chess_board, used_boards, s_n1_tuples, pawn_positions,␣
↪user_wants_pawn, s_n1, s_n1_uniques, s_n1_uniques_tuples):
    tmp_n1, tmp_n1_tuples, tmp_uniques, tmp_uniques_tuples =␣
↪regular_moves(chess_board, used_boards, s_n1_tuples, pawn_positions,␣
↪user_wants_pawn)
    s_n1 += tmp_n1
    s_n1_tuples |= tmp_n1_tuples
    s_n1_uniques += tmp_uniques
    s_n1_uniques_tuples |= tmp_uniques_tuples
```

```
    if user_wants_pawn and chess_board.turn:
        # Push all pawns one row back and check if this leads to new␣
↪boards
        if len(pawn_positions) > 0:
            tmp_list, tmp_set = pawn_moves(chess_board, used_boards,␣
↪s_n1_tuples)
            s_n1 += tmp_list
            s_n1_tuples |= tmp_set
            s_n1_uniques += tmp_list
            s_n1_uniques_tuples |= tmp_set

        # Exchange Queens with Pawns
        queen_positions = check_top_row_for_queen(chess_board)
        if queen_positions:
            tmp_list, tmp_set = replace_queen_with_pawn(chess_board,␣
↪used_boards, s_n1_tuples, queen_positions)
            s_n1 += tmp_list
            s_n1_tuples |= tmp_set
            s_n1_uniques += tmp_list
            s_n1_uniques_tuples |= tmp_set

    return s_n1, s_n1_tuples, s_n1_uniques, s_n1_uniques_tuples
```

Die Funktion `regular_moves` führt für eine übergebene Situation `chess_board` alle `pseudo_legal_moves` durch, um mögliche vorhergehende Situationen zu berechnen. Pseudo-Legale-Züge sind Züge, welche die Figuren auf eine Art bewegen, die der Figur gestattet ist, aber unter Umständen in eine nicht legale Spielsituation führt. Diese werden verwendet, da nur weil der Move von S_{n+1} zu S_n legal ist, der Zug umgekehrt dies nicht sein muss.

Ein simples Beispiel: Situation S_n: Ein König befindet sich ein Feld von einem Schach entfernt.
Diese Position kann erreicht worden sein, da der König von einer Position in S_{n+1} sich aus diesem Schach herausbewegt hat. Der Zug “in das Schach”, wäre jedoch nicht legal, weshalb ein Move aus der Liste der `pseudo_legal_moves` zur Berechnung genommen werden muss. Dies funktioniert nicht für Bauern, da ein Schritt nach “hinten” keine Bewegung ist, welche der Figur zusteht.

Die Funktion überprüft jeden Zug, welcher in der Situation möglich ist. Wenn die

errechnete Situation valide und noch nicht verwendet (überprüft durch Einträge in `used_boards` und `s_n1_tuples`) ist, wird sie den Rückgabe-Variablen angefügt. Wenn sich keine Bauern auf dem Spielfeld befinden (`user_wants_pawn`), dann können die Situationen gespiegelt werden, um weiteren Rechenaufwand zu reduzieren. Diese Spiegelung findet durch eine Iteration über die später definierten Swap_Types statt. Anschließend wird mit der Funktion `mirror` die Spiegelung bestimmt, die Validität der Situation überprüft und ebenfalls an das Ergebnis angefügt.

Nach Abschluss der Berechnungen gibt die Funktion die Liste alle neuen Situationen (ungespiegelt `uniques` und gespiegelt `new_boards`) sowie deren Tupel-Repräsentation wieder.

[ ]:
```
def regular_moves(chess_board, used_boards, s_n1_tuples,␣
↪pawn_positions, user_wants_pawn):
    new_boards = []
    new_tuples = set()
    new_uniques = []
    new_uniques_tuples = set()
    for pLMove in chess_board.pseudo_legal_moves:
        if chess.square_name(pLMove.from_square) not in pawn_positions:

            chess_board.push(pLMove)

            chess_board.turn = chess_board.turn ^ True
            if not chess_board.is_valid() or chess_board.outcome() is␣
↪not None:
                chess_board.turn = chess_board.turn ^ True
                chess_board.pop()
                continue

            # If the new board is found in S, it can be reached in one␣
↪step
            tuple_rep = (chess_board.turn,chess_board.__str__())
            if tuple_rep not in used_boards and tuple_rep not in␣
↪s_n1_tuples and tuple_rep not in new_tuples:
                new_uniques.append(chess_board.copy())
                new_uniques_tuples.add(tuple_rep)

                new_boards.append(chess_board.copy())
```

```
                new_tuples.add(tuple_rep)

                if not user_wants_pawn:
                    for swtype in Swap_Type:
                        mirrored_board = mirror(chess_board, swtype)
                        tuple_rep_mir = (mirrored_board.
↪turn,mirrored_board.__str__())
                        if tuple_rep_mir not in used_boards and␣
↪tuple_rep_mir not in s_n1_tuples and tuple_rep_mir not in new_tuples:
                            new_boards.append(mirrored_board.copy())
                            new_tuples.add(tuple_rep_mir)
            chess_board.turn = chess_board.turn ^ True
            chess_board.pop()

    return new_boards, new_tuples, new_uniques, new_uniques_tuples
```

Wie zuvor bereits erwähnt, ermöglicht die quadratische Natur des Schachbrettes es das Spielbrett zu spiegeln / rotieren und weitere Situationen zu erhalten.

Zunächst wird ein Enum erstellt, welches es ermöglicht über die Arten der Figurenvertauschungen zu iterieren. `Swap_Type` übersetzt zu einem String, welcher im nächsten Schritt als Key für ein Dictionary verwendet wird.

[ ]:
```
class Swap_Type(Enum):
    VERTICAL = "vertical"
    HORIZONTAL = "horizontal"
    ROTATE_RIGHT = "rotate_right"
    ROTATE_180 = "rotate_180"
    ROTATE_LEFT = "rotate_left"
```

Für den Tausch wird über jede Figur iteriert und diese an die entsprechende Position gesetzt. Das Ergebnis wird als `Board-Objekt` zurückgegeben.

Die Formeln zum Spiegeln und Rotieren der Spielsituationen wurden dieser Quelle entnommen.

[ ]:
```
def mirror(board, sw_type : Swap_Type):
    swaps = {
        "vertical" : {x:x^56 for x in range(64)},
        "horizontal" : {x:x^7 for x in range(64)},
```

```
        "rotate_right" : {x:(((x >> 3) | (x << 3)) & 63) ^ 56 for x in␣
↪range(64)},
        "rotate_180" : {x : x ^ 63 for x in range(64)},
        "rotate_left" : {x : (((x >> 3) | (x << 3)) & 63) ^ 7 for x in␣
↪range(64)}
    }

    swapped_board = chess.Board()
    swapped_board.clear()
    swapped_board.turn = board.turn

    for position, piece in board.piece_map().items():
        swapped_board.set_piece_at(swaps[sw_type.value][position],␣
↪piece)

    return swapped_board
```

Befinden sich Bauern in der Situation, müssen diese manuell platziert werden, da für diese auch in den `pseudo_legal_moves` nur die Züge $S_n \rightarrow S_{(n+1)}$ aufgeführt sind. Die Funktion `pawn_moves` erfüllt diese Anforderung. Ähnlich wie die Funktion `regular_moves` werden für eine Situation `chess_board` alle Situationen berechnet, welche durch Bewegung eines Bauerns zu `chess_board` werden. Hierfür wird über alle Bauern auf dem Spielfeld iteriert, diese entfernt und auf das Feld mit dem Index $n-8$ wieder gesetzt. Da eine Reihe 8 Felder hat, hat das Feld in derselben Linie aber vorherigen Reihe den Index 8 geringer. Auch diese Situationen werden sowohl auf Validität als auch bisheriges Vorkommen überprüft, bevor sie den Rückgabevariablen angefügt werden. Das übergebene Objekt wird zu seinem Ursprungszustand zurückgeführt.

[ ]:
```
def pawn_moves(chess_board, used_boards, s_n1_tuples):
    new_boards = []
    new_tuples = set()
    chess_board = chess_board.copy()

    for pawn in chess_board.pieces(chess.PAWN, True):
        if chess_board.piece_at(pawn - 8) is None:
            chess_board.remove_piece_at(pawn)
            chess_board.set_piece_at(pawn - 8, chess.Piece.
↪from_symbol('P'))
```

```
            if chess_board.is_valid() and chess_board.outcome() is None:
                tuple_rep = (chess_board.turn,chess_board.__str__())
                if tuple_rep not in used_boards and tuple_rep not in␣
↪s_n1_tuples:
                    new_boards.append(chess_board.copy())
                    new_tuples.add(tuple_rep)

    return new_boards, new_tuples
```

Ein Problem, das bei der Verwendung der Rückwärts-Analyse auftritt, liegt in dem Szenario: “König und Bauer gegen König”. Dieses Szenario beinhaltet die Umwandlung des Bauerns, welcher die oberste Zeile erreicht hat, in eine andere Figur (Dame, Turm, Läufer, Springer). Da die Dame die stärkste Figur im Spiel ist, wird immer dieser Tausch gewählt. Hat der Nutzer bei den weißen Figuren, welche sich in der Situation sollen, einen Bauern angegeben, wurde dieser beim Errechnen der Menge S_0 durch eine Königin ersetzt.

Für die Berechnung der idealen Züge muss der Bauer wieder in die Situationen, welche sich in den S_n Mengen befinden, eingeführt werden. Der Tausch eines Bauerns zu einer Dame kann nicht durch die `pseudo_legal_moves` umgekehrt werden.

Die Funktion `check_top_row_for_queen` überprüft, ob ein solcher Tausch möglich ist. Sie erhält als Parameter eine Situation `board`, für welches die Felder der obersten Zeile überprüft und jedes zurückgegeben wird, auf dem sich eine Dame befindet.

[ ]:
```
def check_top_row_for_queen(board):
    return_list = []
    for i in range(56, 64):
        if board.piece_type_at(i) == chess.QUEEN:
            return_list.append(i)

    if len(return_list) > 0:
        return return_list
    else:
        return False
```

Wurden mittels der vorhergehenden Funktion Damen in der obersten Zeile gefunden, ersetzt `replace_queen_with_pawn` alle diese Positionen (`toprow_queen_positions`) durch einen Bauern in der vorletzten Zeile. Es wird über die übergebenen Positionen von Damen in der obersten Reihe iteriert, diese entfernt und in der Reihe davor (Feld

Index um 8 verringert) ein Bauer platziert. Wenn die Situation ein valides Schachbrett darstellt, wird sie an die Rückgabeliste angefügt.

```
[ ]: def replace_queen_with_pawn(orig_board, used_boards, s_n1_tuples,␣
     ↪toprow_queen_positions):
        new_boards = []
        new_tuples = set()

        for square in toprow_queen_positions:
            chess_board = orig_board.copy()
            if chess_board.piece_at(square - 8) is None:
                chess_board.remove_piece_at(square)
                chess_board.set_piece_at(square - 8, chess.Piece.
     ↪from_symbol('P'))
                if chess_board.is_valid() and chess_board.outcome() is None:
                    tuple_rep = (chess_board.turn,chess_board.__str__())
                    if tuple_rep not in used_boards and tuple_rep not in␣
     ↪s_n1_tuples:
                        new_boards.append(chess_board.copy())
                        new_tuples.add(tuple_rep)
                chess_board.remove_piece_at(square - 8)
                chess_board.set_piece_at(square, chess.Piece.
     ↪from_symbol('Q'))
        return new_boards, new_tuples
```

Da Bauern mittels der Funktion `pawn_moves` gesondert behandelt werden müssen, muss in `regular_moves` verhindert werden, dass Züge mit Bauern durchgeführt werden. Hierfür wird die Information benötigt, auf welchen Feldern sich ein Bauer befindet. Diese Information wird durch die Funktion `find_pawns` generiert.

```
[ ]: def find_pawns(chess_board):
        result = []
        for pawn in chess_board.pieces(chess.PAWN, True):
                result.append(chess.square_name(pawn))
        return result
```

Wenn mittels der KI eine Spielsituation ausgewertet wird, kann für jeden Zug des weißen Spielers ein Zug ausgewählt werden. Für die Situationen, bei denen Schwarz am Zug ist, muss die KI alle möglichen Züge auswerten können. Da jedoch für einen spezifischen Zug, welcher eine Situation von S_n in S_{n-1} führt, dasselbe nicht für alle

Züge gilt, welche in der Situation möglich sind, müssen die Situationen, bei welchen Schwarz am Zug ist, besonders gefiltert werden. Für jede Situation b aus einem S_n mit $n \,\%\, 2 = 0$ muss folglich gelten:

$$b \in S_n \implies \forall m \in valid\_moves(b) : b.push(m) \in S_m \wedge m < n$$

Wobei `valid_moves` die Liste der legalen Züge für eine Situation ist und `b.push(m)` die Situation beschreibt, welche durch Ausführen des Zuges m entsteht.

Die Funktion `check_black_determinism` stellt dies sicher. Für jede Situation in `s_n1` wird jeder mögliche legale Zug ausgeführt und überprüft, ob die entstehende Situation in einer Menge S_m mit $m \leq n$ auffindbar ist. Nur wenn alle Züge diese Bedingung erfüllen, wird das Objekt in die Liste `s_n1_tmp`, welche in der Funktion `previous_states` die eigentliche Liste `s_n1` ersetzen wird, aufgenommen.

[ ]:
```
def check_black_determinism(s_n1, used_boards, uniques_n1_tuples):
    s_n1_tmp = []
    s_n1_tuples_tmp = set()
    uniques_n1_tmp = []


    for chess_board in s_n1:
        include = True
        if not chess_board.turn:
            for move in chess_board.legal_moves:
                chess_board.push(move)
                tuple_rep = (chess_board.turn,chess_board.__str__())
                if tuple_rep not in used_boards:
                    include = False
                chess_board.pop()
        if include:
            s_n1_tmp.append(chess_board)
            s_n1_tuples_tmp.add((chess_board.turn,chess_board.
↪__str__()))
            tuple_rep = (chess_board.turn,chess_board.__str__())
            if tuple_rep in uniques_n1_tuples:
                uniques_n1_tmp.append(chess_board)

    return s_n1_tmp, s_n1_tuples_tmp, uniques_n1_tmp
```

3.7 Export in Datei

Nach erfolgreicher Berechnung einer S_n Menge werden die FENs der Situationen in eine temporäre `.preConvert` Datei geschrieben. Wurden alle S_n berechnet, wird die temporäre Datei in eine `.chessAI` Datei für die Verwendung in der KI und eine `.chessTest` für das Testen der Ergebnisse konvertiert.

Damit keine Werte einer vergangenen Berechnung in der temporären Datei vorliegen, muss zuerst eine leere `.preConvert` Datei erstellt werden.

[ ]:
```
def create_empty_file(filename):
    f = open("S_n_Results/" + filename + ".preConvert", "w")
    f.write("")
    f.close()
```

Die Zwischenergebnisse werden stetig mittels der `append_to_file` Funktion an die zuvor erstellte `.preConvert` Datei angehängt. Jede Zeile entspricht hierbei einem n aus den S_n Mengen. Für die Zwischenergebnisse werden die FENs als JSON gespeichert.

[ ]:
```
def append_to_file(s_n, filename):
    s_n_ascii = []
    for board in s_n:
        s_n_ascii.append(board.fen())

    f = open("S_n_Results/" + filename + ".preConvert", "a")
    f.write("\n")
    f.write(json.dumps(s_n_ascii))
    f.close()
```

Nachdem die gesamte Sequenz aller n berechnet wurde, muss die temporäre Datei in zwei Dateien zur Auswertung konvertiert werden. Die `.chessAI` Datei enthält Mengen von Tupeln. Die `.chessTest` Datei enthält die FENs, um in den Test-Szenarien wieder Board-Objekte erstellen zu können.

Zuerst müssen die Informationen über die berechneten S_n Mengen aus der `.preConvert` Datei gelesen werden. Die darin gespeicherten FENs werden in einer Liste gespeichert und zusätzlich zu Board-Objekten instanziiert. Die Objekte werden in die zuvor bereits verwendete Tupel-Darstellung gewandelt und in eine Menge eingefügt.

Zum Speichern der Dateien wird aus Effizienzgründen das Modul `pickle` verwendet, welches die Daten in Binärdateien speichert. Da die `.chessAI` Dateien gegebenen-

falls an Nutzer der KI verteilt werden müssen, werden diese zusätzlich mit dem Modul `ZipFile` komprimiert. Die `.chessTest` Dateien werden nur zum Evaluieren der Ergebnisse verwendet und nicht an Nutzer verteilt. Damit sie schneller eingelesen werden können, werden sie nicht komprimiert.

[ ]:
```
def convert_file(filename):
    s_n_seq_fens = []
    s_n_seq_tuples = []
    f = open("S_n_Results/" + filename + ".preConvert", "rb")
    lines = f.readlines()
    first = True
    for line in lines:
        # First line is empty
        if first:
            first = False
            continue

        tmp_list = []
        tmp_set = set()
        tmp = json.loads(line)

        for fen in tmp:
            tmp_list.append(fen)
            tmp_board = chess.Board(fen)
            tmp_set.add((tmp_board.turn,tmp_board.__str__()))
        s_n_seq_fens.append(tmp_list)
        s_n_seq_tuples.append(tmp_set)
    f.close()

    f = open("S_n_Results/" + filename + ".pickle", "wb")
    f.write(pickle.dumps(s_n_seq_tuples))
    f.close()

    f = open("S_n_Results/" + filename + ".chessTest", "wb")
    f.write(pickle.dumps(s_n_seq_fens))
    f.close()

    with ZipFile("S_n_Results/" + filename + '.chessAI', 'w',␣
↪compression=ZIP_DEFLATED) as zipped:
```

```
        zipped.write("S_n_Results/" + filename + ".pickle", filename+".
↪pickle")
    if os.path.exists("S_n_Results/" + filename + ".chessAI") and os.
↪path.exists("S_n_Results/" + filename + ".pickle"):
        os.remove("S_n_Results/" + filename + ".pickle")

    if os.path.exists("S_n_Results/" + filename + ".preConvert"):
        os.remove("S_n_Results/" + filename + ".preConvert")
```

Mit den zuvor definierten Funktionen ist es nun möglich, alle S_n zu bestimmen. Hierfür wird die Funktion `previous_states` solange aufgerufen, bis keine Situationen für ein $n+1$ mehr gefunden werden und die Liste S_{n+1} leer ist.

Die Funktion `run` geht hierbei von einem `s_n`, welches zuvor bestimmt wurde, aus und durchläuft diesen Prozess in einer Schleife. Nach abgeschlossener Rechnung wird das Konvertieren der Datei gestartet.

[ ]:
```
def run(s_n, used_boards, n, user_wants_pawn, filename, uniques):
    if n == 0:
        append_to_file(s_n, filename)

    while True:
        n += 1
        s_n, used_boards, uniques = previous_states(used_boards, n,␣
↪user_wants_pawn, uniques)
        print("S " + str(n) + ": " + str(len(s_n)))
        if not s_n: #an empty list is false
            break
        append_to_file(s_n, filename)
        del s_n

    print("Done")
    print(str(n) + " S-Lists calculated")

    convert_file(filename)

    print("File converted")
```

Die Funktion `run_from_start` berechnet S_0 und startet anschließend die Berechnung

aller S_n.

```
[ ]: def run_from_start(user_supplied_pieces, filename):
         user_wants_pawn = chess.Piece.from_symbol("P") in␣
     ↪user_supplied_pieces

         s_0, used_boards, uniques = setup_boards(user_supplied_pieces,␣
     ↪user_wants_pawn)
         create_empty_file(filename)
         run(s_0, used_boards, 0, user_wants_pawn, filename, uniques)
```

Wurde die Berechnung abgebrochen (z.B. durch einen Neustart des Computers) kann die Funktion `resume_from_file` den Zustand, welcher in einer `.preConvert` Datei gespeichert wurde wiederherstellen, und die Berechnung beim nächsten n fortsetzen. Hierfür werden alle bereits berechneten n in Objekte instanziiert und die Tupel Repräsentationen als bereits verwendete Situationen (`used_boards`) gespeichert. Für das höchste n, werden die Objekte behalten und der `run` Funktion übergeben.

```
[ ]: def resume_from_file(filename):
         used_boards = set()
         s_n = []
         count = -1

         f = open("S_n_Results/" + filename + ".preConvert", "r")
         lines = f.readlines()
         first = True
         for line in lines:
             # First line is empty
             if first:
                 first = False
                 continue

             tmp_list = []
             tmp_set = set()
             tmp = json.loads(line)

             for fen in tmp:
                 chess_board = chess.Board(fen)
                 tmp_list.append(chess_board)
```

```
            tmp_set.add((chess_board.turn,chess_board.__str__()))
        s_n = tmp_list
        used_boards |= tmp_set
        count += 1
    f.close()
    print("Starting at S" + str(count) + " (" + str(len(s_n)) + "␣
↪Boards)")

    # Adding a few Pawns to the result only slightly increases the␣
↪file_size, and the value of user_wants_pawn is \
    # currently not stored
    run(s_n, used_boards, count, True, filename, s_n)
```

3.8 Konfigurations Variablen

Für die Bestimmung der gewonnenen Spielbretter müssen nun die Spielfiguren angegeben werden, für die die Endspiel-Datenbank berechnet werden soll. Hierfür werden in der Liste `pieces_to_place` alle Figuren aufgeführt. Die Einträge der Liste werden als Tupel bestehend aus Figur und Farbe gespeichert. Bsp.: `(chess.KING, chess.WHITE)` Die Reihenfolge oder Position der Figur ist für die Berechnung irrelevant. Diese wird erst im nächsten Schritt (der Auswertung) benötigt.

Der `FILENAME` wird für das Speichern der Ergebnisse verwendet.

[ ]:
```
WHITE_PIECES_TO_PLACE = [chess.Piece.from_symbol('P')]

FILENAME = "S_n_seq_pawn"
```

3.9 Start der Rechnung

[ ]:
```
%%time
run_from_start(WHITE_PIECES_TO_PLACE, FILENAME)
#resume_from_file(FILENAME)
```

4 Spielen gegen die KI

4.1 Vom Nutzer zu tätigende Einstellungen

Zur Erstellung eines individuellen Spielfeldes werden Dictionaries verwendet. Diese verwenden jeweils als `key` die Figur, die auf dem Spielfeld platziert werden soll. Als `value` nutzen die Einträge jeweils eine Liste, die vom Nutzer mit den String-Bezeichnern der Felder gefüllt werden sollen, auf denen die jeweilige Figur steht. Die Bezeichner folgen hierbei dem Format, welches in Notebook `01_chess_introduction` erklärt wurde.

[ ]:
```
WHITE_POSITIONS = {chess.KING:['e1'],
                   chess.QUEEN:['a6'],
                   chess.ROOK:[],
                   chess.BISHOP:[],
                   chess.KNIGHT:[],
                   chess.PAWN:[]}

BLACK_POSITIONS = {chess.KING:['e7'],
                   chess.QUEEN:[],
                   chess.ROOK:[],
                   chess.BISHOP:[],
                   chess.KNIGHT:[],
                   chess.PAWN:[]}
```

Weiter muss der Nutzer angeben, welche vorher berechnete Spielsituation er laden möchte. Hierfür muss der Dateiname in einer globalen Variable (ohne Dateiendung `.chessAI`) angegeben werden.

[ ]:
```
FILE = "S_n_seq_queen"
```

4.2 Logik für die Interaktion mit der Spielsituation

Im folgenden Abschnitt sollen einige Hilfsfunktionen erklärt werden, welche für die Interaktion zwischen dem Nutzer über ein Jupyter Notebook und der Spielsituation benötigt werden.

Die Funktion `get_occupied_cells()` übersetzt die zuvor erstellten und vom Nutzer veränderten `Position-Dictionaries` in eine Liste, die alle `values` der Dictionaries, also besetzte Felder enthält. Diese Liste wird benötigt, um zu überprüfen, ob die Eingaben des Nutzers korrekte Zellen sind.

```
[ ]: def get_occupied_cells():
         cells = []
         for values in WHITE_POSITIONS.values():
             for value in values:
                 cells.append(value)
         for values in BLACK_POSITIONS.values():
             for value in values:
                 cells.append(value)
         return cells
```

Mit der Information von der vorherigen Funktion übernimmt die Funktion `check_for_correct_cells()` die Überprüfung. Hierzu wird jeder Wert in den eingegebenen Feldern in den Dictionaries `WHITE_POSITIONS` und `BLACK_POSITIONS` auf Einhaltung eines regulären Ausdrucks, welcher auf eine Kombination aus Buchstabe und Zahl überprüft. Das Ergebnis wird in Form eines booleschen Werts zurückgegeben.

```
[ ]: def check_for_correct_cells():
         cells = get_occupied_cells()
         for cell in cells:
             x = re.search("[a-h][1-8]", cell)
             if x is None:
                 print(cell)
                 print("Value incorrect!")
                 return False
             else:
                 pass
         return True
```

Bevor die Figuren dem Objekt hinzugefügt werden, werden sie in einem Dictionary gesammelt, welches als `Piece-Map` verwendet wird.

Die Funktion `collect_cells(color, pieces)` erstellt aus einem Booleschen-Wert für die Spielerfarbe und einem Dictionary mit Figuren und Spielfeldern eine Liste mit `Piece` Objekten, welche als Index den Index des Spielfeldes verwendet. Diese Liste entspricht dem Format, mit welchem Formationen von der Bibliothek geladen werden können.

```
[ ]: def collect_cells(color, pieces):
         occupied_cells = {}
         for piece_type, values in pieces:
```

```
            piece = chess.Piece(piece_type, color)
            for value in values:
                square = chess.parse_square(value)
                occupied_cells[square] = piece

    return occupied_cells
```

Die Funktion `create_board()` füllt das Schachbrett letztlich mit den Figuren, die in den Dictionaries angegeben wurden. Als Rückgabewert gibt die Funktion das gefüllte Schachbrett als Objekt der `chess` Library zurück. Hierzu werden die vorherigen Funktionen verwendet, um die `Piece-Map` zu erstellen, diese auf das Board-Objekt angewandt und auf Validität überprüft. Ist das Board nicht valide, wird ein Fehler ausgegeben.

Ein Problem, welches in Spielsituationen mit dem Turm auftritt, ist die Berechtigung zu einer Rochade. Damit das Board als valide anerkannt wird, werden die entsprechenden Flags mit der Funktion `clean_castling_rights` korrekt gesetzt.

[ ]:
```
def create_board():
    local_board = chess.Board()
    occupied_cells = {}
    if check_for_correct_cells():
        occupied_cells |= collect_cells(chess.WHITE, WHITE_POSITIONS.
↪items())
        occupied_cells |= collect_cells(chess.BLACK, BLACK_POSITIONS.
↪items())

        local_board.set_piece_map(occupied_cells)
        local_board.castling_rights = local_board.
↪clean_castling_rights()

        # Herausfinden, weshalb momentan angegebenes Board invalid ist
        if not local_board.is_valid():
            display(local_board)
            print("Specified lineup is invalid")
            print("Default board created instead")
            local_board = chess.Board()
    return local_board
```

4.3 Import der Daten

Für die Bestimmung der Züge der KI werden die S_n Mengen verwendet, die im Notebook `calculation.ipynb` berechnet werden.
Eine Erklärung worum es sich hierbei handelt, findet sich in diesem Notebook. Die Ergebnisse der Berechnung werden mittels `pickle` serialisiert und in einer ZIP-Datei komprimiert abgespeichert. Zur Verwendung wird das Archiv entpackt und die Liste deserialisiert. Weitere Informationen zum Inhalt der Datei befinden sich ebenfalls im Notebook `calculation.ipynb`.

Die Funktion `load_s_n_sequence(filename)` erhält den Dateinamen (`filename`) als Parameter und gibt die Endspieldaten zurück.

[ ]:
```
def load_s_n_sequence(filename):
    s_n_sequence_tuples = []
    with ZipFile("S_n_Results/" + filename + ".chessAI") as zipped:
        with zipped.open(filename + ".pickle") as calculation:
            tmp = pickle.loads(calculation.read())
            for item in tmp:
                s_n_sequence_tuples.append(item)
    return s_n_sequence_tuples
```

4.4 Den besten Zug für die KI ermitteln

Um mit den berechneten Endspieldaten gegen einen Spieler zu gewinnen, muss jeder Zug, den Weiß macht, optimal sein. Ein passender Zug für die KI besteht darin die Situation von S_n in einen Zustand zu überführen, in dem sie sich in S_{n-1} befindet.

Die Funktion `find_next_move(curr_board, s_index, s_n_sequence)` bestimmt für ein übergebenes Board-Objekt einen solchen Spielzug. Aus Effizienzgründen wird zusätzlich zum gefundenen Spielzug (`move`) der neue Wert für n zurückgegeben.

Für diese Berechnung benötigt `find_next_move` nachkommende Argumente:

- `curr_board`: Das Board, für welches der nächste Spielzug berechnet werden soll.
- `s_index`: Das n eines S_n, in welchem sich `curr_board` befindet. Wird aus Effizienzgründen übergeben.
- `s_n_sequence`: Die Liste mit allen S_n.

Kann kein Spielzug gefunden werden, gibt die Funktion den Wert -1 zurück.

Die für diesen Ablauf benötigte Funktion `find_situation_in_sequence` wird im Notebook `functions.ipynb` definiert und erklärt.

[ ]:
```
def find_next_move(curr_board, s_index, s_n_sequence):
    s_index_new = s_index - 1
    for move in curr_board.legal_moves:
        curr_board.push(move)
        curr_tupel = (curr_board.turn, curr_board.__str__())
        _tmp = find_situation_in_sequence(curr_tupel,␣
 ↪[s_n_sequence[s_index_new]])
        if _tmp != -1:
            curr_board.pop()
            return s_index_new, move
        curr_board.pop()

    return -1, None
```

4.5 Globale Variablen für die Anzeige

Nachdem alle Funktionen für die Bestimmung eines Zugs definiert worden sind, gilt es die bereitgestellte UI der `python-chess` Bibliothek zu erweitern. In Form von globalen Variablen werden UI-Elemente definiert, die für die Eingabe eines neuen Zugs benötigt werden:

- `input_field` = Ein Eingabefeld, in dem der nächste Zug von Weiß eingetragen werden soll.
- `execute_button` = Ein Button, der nach der Auswahl des Zuges diesen auch auf dem Schachbrett ausführt.

[ ]:
```
input_field = widgets.Text()

execute_button = widgets.Button(
    description='Execute Move',
    disabled=False,
    button_style='',
    tooltip='Executes the move selected with the dropdowns "Piece:" and␣
 ↪"Move:"',
    icon='check'
)
```

4.6 Globale Variablen für den aktuellen Spielzustand

Damit der aktuelle Stand des Spiels auch ohne eine lineare Kette von Funktionsaufrufen verfügbar ist, werden diese Informationen in globalen Variablen gespeichert.

[ ]:
```
S_N_TUPLES = load_s_n_sequence(FILE)
S_INDEX = 0
BOARD_INDEX = 0
```

Weiterhin werden weitere globale Variablen definiert, die einerseits den Dateinamen für die Spielzüge (`filename`), andererseits das Spielbrett zum Spielen gegen die KI innehalten (`board`).

Innerhalb der python-chess library werden die Farben des Schachbretts `chess.WHITE` und `chess.BLACK` als boolesche Variablen definiert. Deswegen wurde zur späteren Dokumentation, aber auch zum Debuggen eine Funktion geschrieben, die die Farbe, die gerade am Zug ist, als String zurückgibt.

[ ]:
```
def get_color(turn):
    if turn:
        return "White"
    else:
        return "Black"
```

Zum Definieren der globalen Variablen `filename` wird die Funktion `update_filename` definiert.

[ ]:
```
def update_filename(new_value):
    global FILENAME
    FILENAME = new_value
```

Weiter werden für den Spielbeginn (`reset_board`) und auch für ausgeführte Spielzüge (`reset_input_field`) Funktionen geschrieben, die die UI Elemente auf ihren Standardwert zurücksetzen.

[ ]:
```
def reset_input_field():
    global input_field
    input_field.value = ''
```

[ ]:
```
def reset_board():
    global BOARD
    BOARD = create_board()
```

```
    global S_INDEX
    global BOARD_INDEX
    tupel = (BOARD.turn, BOARD.__str__())
    S_INDEX = find_situation_in_sequence(tupel, S_N_TUPLES)
    if BOARD.turn:
        S_INDEX, move = find_next_move(BOARD, S_INDEX, S_N_TUPLES)
        if S_INDEX == -1:
            print("No Move for white found")
        else:
            execute_move(move, BOARD.turn)
            reload_screen()
```

Zum Durchführen der Spielzüge im Board-Objekt wird die Hilfsfunktion `execute_move` verwendet. Die Funktion erhält als Parameter einen `chess.Move` und führt diesen auf dem globalen `BOARD` für die Farbe `chess.turn` durch. Zusätzlich wird in der Datei, die in unter dem Namen `filename` zu finden ist, für den Zug ein Eintrag hinterlegt.

[ ]:
```
def execute_move(move, turn):
    global BOARD
    global S_INDEX
    global BOARD_INDEX
    move_file = open("Played_Games/" + FILENAME, "a")
    if turn:
        move_file.write(str(BOARD.fullmove_number) + ". " + move.uci()␣
↪+ " ")
    else:
        move_file.write(move.uci() + "\n")
    move_file.close()
    BOARD.push(move)
    tupel = (BOARD.turn, BOARD.__str__())
    S_INDEX = find_situation_in_sequence(tupel, S_N_TUPLES)
```

4.7 Weitere Hilfsfunktionen

Das Ziel der Funktion `get_pieces_placed_on_board` besteht darin, von einer mitgegebenen `chess.Color` die Figuren zu bestimmen, die noch auf dem Schachbrett stehen. Zurück gibt sie ein Dictionary, das als `key` die Schachfiguren und als `value` eine Liste mit den Positionen der Figuren besitzt.

[ ]:
```
def get_pieces_placed_on_board(color):
    piece_type_to_string = {
        1 : "Pawn",
        2 : "Knight",
        3 : "Bishop",
        4 : "Rook",
        5 : "Queen",
        6 : "King"
    }
    pieces_with_position = {
        "Pawn" : [],
        "Knight" : [],
        "Bishop" : [],
        "Rook" : [],
        "Queen" : [],
        "King" : []
    }
    for piece_square, color_piece in BOARD.piece_map().items():
        if color_piece.color == color:
            pieces_with_position[piece_type_to_string[color_piece.
↪piece_type]].append(chess.square_name(piece_square))
    return pieces_with_position
```

Bei der Funktion `get_moves_from_square` werden anhand der mitgegebenen `legal_moves` für ein vorgegebenes Feld die Züge selektiert, die man von diesem Feld aus ziehen kann.

[ ]:
```
def get_moves_from_square(square, legal_moves):
    moves = []
    for possible_move in legal_moves:
        if square == chess.square_name(possible_move.from_square):
            moves.append(possible_move)
    return moves
```

Da die Anzeige des UI auf Konsolenausgaben basiert, muss diese auch nach einem neuen Zug geleert werden, sodass die Konsole nicht mit den Elementen überflutet wird. Hierfür aktualisiert die Funktion `reload_screen` die Ausgabe und zeigt erneut das Schachbrett, die beiden Dropdowns und den Knopf zum Ausführen des Zugs an.

[ ]:
```
def reload_screen():
    clear_output()
    display(BOARD, input_field, execute_button)
    display(Javascript("setTimeout(function focus() {document.
 ↪querySelector('input').focus()}, 100);"))
```

Die Funktion `show_end_screen` hingegen zeigt nur das Schachbrett an.

[ ]:
```
def show_end_screen():
    clear_output()
    display(BOARD)
```

Die `game_result(result)` Funktion gibt den Grund, weshalb das Spiel beendet wurde zurück.

[ ]:
```
def game_result(result):
    if result == Termination.CHECKMATE:
        return get_color(BOARD.turn) + " has lost because of Checkmate!"
    elif result == Termination.STALEMATE:
        return "It's a draw!"
    elif result == Termination.INSUFFICIENT_MATERIAL:
        return "No side can win the game anymore!"
    elif result == Termination.SEVENTYFIVE_MOVES:
        return "The game is drawn because half-move clock is greater␣
 ↪than 150 since a capture or a pwn has been moved."
    elif result == Termination.FIVEFOLD_REPETITION:
        return "The game is drawn because the current position occurred␣
 ↪the fifth time!"
    elif result == Termination.FIFTY_MOVES:
        return "The game is drawn because half-move clock is greater␣
 ↪than 100 since a capture or a pwn has been moved."
    elif result == Termination.THREEFOLD_REPETITION:
        return "The game is drawn because the current position occurred␣
 ↪the third time!"
    elif result == Termination.VARIANT_WIN:
        return get_color(BOARD.turn) + " has won because of␣
 ↪variant-specific conditions"
    elif result == Termination.VARIANT_LOSS:
```

```
        return get_color(BOARD.turn) + " has lost because of␣
↪variant-specific conditions"
    elif result == Termination.VARIANT_DRAW:
        return "Game is drawn because of variant-specific conditions!"
    else:
        return "Something went wrong!"
```

Für den `execute_button` und das `input_field` wird eine Funktion geschrieben, die für das Ausführen des Zuges verantwortlich ist. Diese wird entweder durch Klicken des Buttons oder Drücken der Enter-Taste aufgerufen. Sobald der eingegebene Zug ausgeführt wurde, wird mithilfe der eingelesenen S_n Mengen der Zug für die KI bestimmt. Weiterhin wird auch in dieser Funktion überprüft, ob das Spiel bereits beendet wurde.

[ ]:
```
def execute_entered_move(change):
    global BOARD
    global S_INDEX
    global BOARD_INDEX
    try:
        if input_field.value != '':
            black_move = chess.Move.from_uci(input_field.value)
            if black_move in list(BOARD.legal_moves):
                execute_move(black_move, BOARD.turn)
                reset_input_field()
                # Next AI move executed by white
                S_INDEX, next_move = find_next_move(BOARD, S_INDEX,␣
↪S_N_TUPLES)
                if S_INDEX == -1:
                    print("No Move for white found")
                else:
                    execute_move(next_move, BOARD.turn)
                    reload_screen()
                if BOARD.legal_moves.count() == 0:
                    # If wanted add different endings
                    show_end_screen()
                    result = BOARD.outcome().termination
                    print(game_result(result))
            else:
                print("Entered a wrong move. Please try again!")
```

```
                print(input_field.value)
                time.sleep(2)
                reload_screen()
        else:
            print("Enter a move!")
            time.sleep(2)
            reload_screen()
    except ValueError:
        print("Entered a wrong move. Please try again!")
        print(input_field.value)
        time.sleep(2)
        reload_screen()
```

Damit die Funktion beim Klicken des `execute_button` oder durch Drücken der Enter-Taste ausgeführt wird, muss diese den `on_click` Events zugewiesen werden.

[ ]:
```
execute_button.on_click(execute_entered_move)
input_field.on_submit(execute_entered_move)
```

Die Funktion `start_game` bereitet alle Parameter für einen Spielverlauf vor. Dazu gehört:

- Erstellung eines neuen Boards.
- Das Erstellen einer neuen Historie für das neu begonnene Spiel.
- Anzeigen der neuen Spielsituation.

[ ]:
```
def start_game():
    global BOARD
    BOARD = create_board()
    update_filename("Move-History_" + str(datetime.today().
 ↪replace(microsecond=0)).replace(":","_") + ".txt")
    move_file = open("Played_Games/" + FILENAME, "a")
    move_file.write(BOARD.fen() + "\n")
    move_file.close()
    reset_board()
```

Mit dem Aufruf der Funktion `start_game` kann nun ein Spiel gegen die KI gestartet werden.

[ ]:
```
start_game()
```

5 Wiedergeben einer bereits gespielten Partie

In diesem Bereich werden Funktionen definiert, die das Wiedergeben einer Partie ermöglichen. Hierfür wird eine Spiel-Historie eingelesen. Historien werden bei der Ausführung des `03_play_against_ai.ipynb` Notebooks erstellt und in dem Ordner `Played_Games` hinterlegt. Am Ende dieses Notebook wird der Spielverlauf mit einem selbst festgelegten Tempo der Züge dargestellt.

5.1 Importieren des Spielverlaufs

Als ersten Schritt muss eine beliebige Schach-Historie eingelesen werden. Hierfür wird die Funktion `get_lines_from_file` definiert, die für einen mitgegebenen relativen Pfad zu einer Schach-Historie (`file_name`) die Zeilen der Datei ohne Zeilenumbruch (`"\n"`) zurückgibt.

[ ]:
```
def get_lines_from_file(file_name):
    move_file = open(file_name, "r")
    all_lines = move_file.readlines()

    # Remove '\n' from the lines
    lines = [all_lines[x][:-1] for x in range(len(all_lines)) if␣
 ↪len(all_lines[x]) > 12]
    if len(all_lines) == len(lines) + 1:
        lines += [all_lines[-1]]
    return lines
```

5.2 Auslesen der Startposition und Züge

Um den ausgewählten Spielverlauf darstellen zu können, muss ein neues Spielbrett erstellt werden.

[ ]:
```
BOARD = chess.Board()
BOARD
```

Mit der Funktion `set_board` wird für einen relativen Pfad zu einer Spielhistorie(`file_name`) der Zustand in der globalen Variable `BOARD` dargestellt, der zu Beginn der Datei vorherrschte. Dieser kann in der ersten Zeile der Datei vorgefunden werden.

```
[ ]: def set_board(file_name):
         global BOARD
         lines = get_lines_from_file(file_name)
         BOARD.set_fen(lines[0])
```

Um den Spielverlauf darstellen zu können, werden die Spielzüge benötigt. Diese können mit der Funktion `get_moves_from_file` ausgewertet werden. Hierfür wird ebenfalls der relative Pfad zu einer Spielhistorie (`file_name`) mitgegeben.

```
[ ]: def get_moves_from_file(file_name):
         lines = get_lines_from_file(file_name)
         moves = []
         for line in lines[1:]:
             split_move = line.split(' ')
             moves.append(split_move[1:])
         return moves
```

5.3 Spiel wiederholen

Die Wiederholung des Spiels kann nun mit der Ausführung der Funktion `start_replay` begonnen werden. Diese benötigt als Parameter den relativen Pfad zu einer Spielhistorie (`file_name`) und zusätzlich eine Anzahl an Sekunden, die nach der Ausführung eines Zuges gewartet werden sollen (`delay`). Das Spiel wird daraufhin in der Konsole für den Nutzer schrittweise angezeigt.

```
[ ]: def start_replay(file_name, delay):
         set_board(file_name)
         moves = get_moves_from_file(file_name)
         display(board)
         time.sleep(2)
         for move_pair in moves:
             for move in move_pair:
                 if move == '':
                     print("Finished!")
                 else:
                     clear_output()
                     board.push(chess.Move.from_uci(move))
                     display(board)
                     time.sleep(delay)
```

```
[ ]: start_replay("Played_Games/Move-History_2022-03-11 20_39_21.txt", 1)
```

6 Auswerten eines Schach-Endspiels

Neben der Berechnung der Endspiel-Situationen gilt es zu überprüfen, ob es sich bei den Zügen der KI auch um optimale Züge handelt. Hierfür werden im Rahmen von diesem Notebook Testszenarien geschrieben, welche die Entscheidungen der KI bewerten. Zum Vergleich wurde die Stockfish-Engine herangezogen. Stockfish ist die momentan (Stand: 2022) beste Schach-Engine, die frei zur Verfügung steht (Quelle: chess.com o. D.).

Insgesamt können vier unterschiedliche Szenarien in dem Notebook getestet werden. Die Struktur der Tests ist wie folgt aufgebaut:

- `compare_fen_stockfish`: Dieser Test überprüft für eine mitgegebene FEN jeweils, wie viele Züge die Endspiel-Tabellen und Stockfish zum Gewinnen brauchen.
- `test_random_boards`: Dieser Test überprüft für eine gegebene Anzahl n zufälliger Situationen, ob Stockfish bessere Ergebnisse liefert.
- `compare_sequence_stockfish`: Dieser Test überprüft für ein mitgegebenes n alle Situationen, die sich in der Menge S_n befinden, ob Stockfish oder die KI bessere Ergebnisse erzielt.
- `compare_all_sequences`: Dieser Test überprüft für jegliche Situationen, die einer S_n Menge zugeordnet wurden, ob Stockfish eine Lösung in weniger Zügen bestimmen kann.

Die Ergebnisse der letzten drei Tests werden im Ordner `/Tests` gespeichert.

6.1 Konfiguration

Zu Beginn müssen für den Vergleich unterschiedliche Variablen definiert werden. Diese beinhalten den Namen der `.chessTest` Datei zu den berechneten S_n Sequenzen und einen Pfad zur Stockfish-Installation. Weiterhin wird eine globale Variable `VERBOSE` festgelegt, die das Anzeigen von zusätzlichen Ausgaben ein oder ausschaltet.

```
[ ]: STOCKFISH_PATH = "./stockfish/stockfish.exe"
     VERBOSE = False
```

```
STOCKFISH = stockfish.Stockfish(STOCKFISH_PATH)
```

[ ]:
```
S_N_FILE = "S_n_seq_queen"

S_N_Sequence_fen, S_N_Sequence_fen_short = load_s_n_fens(S_N_FILE)
```

6.2 Vergleich zwischen Stockfish und KI

Für den Vergleich einer Situation wird die Anzahl der Züge zwischen Stockfish und der selbst geschriebenen KI miteinander verglichen. Eine geringere Anzahl ist hierbei besser.

6.2.1 Berechnung der Zuglisten

Um die Anzahl der benötigten Züge zu bestimmen, muss eine Liste angelegt werden, welche Züge durchgeführt werden. Dafür muss immer ein optimaler Zug bestimmt werden. Diese Aufgabe wird von der Funktion `find_next_move` erfüllt. Diese berechnet für eine übergebene FEN (`fen`) den idealen Spielzug. Für die Berechnung wird der `s_index` mitgegeben, in dem die FEN gefunden werden kann. Diese wird für die Suche in den "gekürzten" übergebenen S_n Mengen (`s_n_sequence_short`) verwendet. Die Suche findet mithilfe der bereits definierten Funktion `find_situation_in_sequence` statt.

Die Züge werden wie folgt bestimmt:

- `Stockfish`: Für sowohl Schwarz als auch für Weiß, werden die Züge mit der Funktion `Stockfish().get_best_move()` bestimmt.
- `KI`: Für Weiß werden die Züge anhand der S_n Mengen bestimmt, die schwarzen Züge werden mit der Funktion `Stockfish().get_best_move()` gewählt.

Ein gegnerischer Spieler macht nicht immer einen im Sinne der KI optimalen Zug, und das Auswählen zufälliger Züge entspräche einem unerfahrenen Spieler. Um die Effizienz der KI gegen einen guten Spieler zu demonstrieren werden die Züge von Schwarz beim Bestimmen der Zuganzahl von Stockfish übernommen. Aus demselben Grund spielt Stockfish beim Bestimmen der Engine-Anzahl an Zügen in der folgenden Funktion `stockfish_movelist` gegen sich selbst.

Die Funktion liefert als Rückgabewert den besten Zug `move` und der neue `s_index`, welcher beschreibt, in welcher S_n Menge die Spielsituation zu finden ist.

[ ]:
```
def find_next_move(fen, s_index, s_n_sequence_short):
    STOCKFISH.set_fen_position(fen)
    curr_board = chess.Board(fen)
    if curr_board.turn:
        if VERBOSE:
            print("---White:---")
            print("Starting in S" + str(s_index))
        for move in curr_board.legal_moves:
            curr_board.push(move)
            cur_fen = curr_board.fen()
            _tmp =␣
↪find_situation_in_sequence(get_board_and_turn(cur_fen),␣
↪[s_n_sequence_short[s_index - 1]])
            s_index_tmp = s_index - 1
            if _tmp != -1:
                if VERBOSE:
                    print("    Move: " + str(move))
                    print("    S" + str(s_index_tmp))
                    print("Ended in S" + str(s_index))
                curr_board.pop()
                return s_index_tmp, move
            curr_board.pop()

        return -1, None
    else:
        if VERBOSE:
            print("---Black:---")
            print("Starting in S" + str(s_index))
        move = chess.Move.from_uci(STOCKFISH.get_best_move())
        curr_board.push(move)
        cur_fen = curr_board.fen()
        s_index =␣
↪find_situation_in_sequence(get_board_and_turn(cur_fen),␣
↪s_n_sequence_short[:s_index])
        if VERBOSE:
            print("    Move: " + str(move))
            print("    S" + str(s_index))
            print("Ended in S" + str(s_index))
```

```
        curr_board.pop()
        return s_index, move
```

Bei der Berechnung der Züge für die KI wird die Funktion `calculate_all_moves` verwendet. Diese berechnet in einer Schleife alle Moves ausgehend von einer FEN, bis die KI gewonnen hat. Die Berechnung verwendet hierzu die FEN der Spielsituation (`fen`) und die Liste der gekürzten S_n Mengen (`s_n_sequence_short`). Mithilfe von diesen Parametern wird eine Liste von Zügen bestimmt, in der die Partie zwischen Stockfish und der selbst geschriebenen KI beendet wurde.

[ ]:
```
def calculate_all_moves(fen, s_n_sequence_short):
    moves = []

    s_index = find_situation_in_sequence(get_board_and_turn(fen),␣
 ↪s_n_sequence_short)

    board = chess.Board(fen)
    while s_index > 0:
        cur_fen = board.fen()
        s_index, next_move = find_next_move(cur_fen, s_index,␣
 ↪s_n_sequence_short)
        board.push(next_move)
        moves.append(next_move)

    if s_index == -1:
        return None

    return moves
```

Nachdem eine Liste aller Züge für die KI bestimmt wurden, gilt dies gleichermaßen für die Berechnung von Stockfish umzusetzen. Hierfür erhält die Funktion `stockfish_movelist` eine `fen` mit der aktuellen Spielsituation. In der Funktion bestimmt Stockfish solange die bestmöglichen Züge, bis die Partie beendet worden ist. Diese Liste von Zügen wird am Ende der Funktion zurückgegeben.

[ ]:
```
def stockfish_movelist(fen):
    moves = []

    board = chess.Board(fen)
```

```
    while not board.is_game_over():
        STOCKFISH.set_fen_position(board.fen())
        next_move = chess.Move.from_uci(STOCKFISH.get_best_move())
        board.push(next_move)
        moves.append(next_move)

    return moves
```

6.2.2 Hilfsfunktionen für den Vergleich

Der Vergleich zwischen der Stockfish Engine und der KI basiert auf der Anzahl der Züge, bis Weiß das Spiel gewonnen hat. Hierfür werden drei Hilfsfunktionen definiert, die für den Vergleich benötigt werden.

Bei der ersten Funktion handelt es sich um `compare_move_list`. Diese erhält eine Liste von 3-Tupeln(`move_count_list`), die als ersten Wert die Anzahl der Züge der KI beinhaltet und als zweiten Wert die Anzahl der Züge, die Stockfish zum Beenden der Partie benötigt hat. Der dritte Wert stellt die Differenz zwischen den beiden Anzahlen dar.

Als Ergebnis liefert die Funktion drei Werte zurück, die ein erstes Abbild für die Performanz der KI darstellen. Der erste Wert ist die Anzahl der Spielsituationen, bei denen Stockfish und die KI gleich viele Züge benötigt haben (`equal`). Der zweite Wert ist die Anzahl der Spiele, bei denen die KI weniger Züge benötigt hat (`ki_better`) und der letzte Werte die Anzahl der Spiele, in der Stockish weniger Züge benötigt hat (`stockfish_better`).

[ ]:
```
def compare_move_lists(move_count_list):
    equal = 0
    ki_better = 0
    stockfish_better = 0
    for ki_move, stock_move, diff in move_count_list:
        if diff == 0:
            equal += 1
        elif diff < 0:
            ki_better += 1
        else:
            stockfish_better += 1
    return equal, ki_better, stockfish_better
```

Diese Werte können nun weiter genutzt werden, indem neben der Anzahl der Ergebnisse auch der Grad des Unterschieds bestimmt wird. Hierfür wurde die Funktion `get_average_difference` definiert. Diese nutzt ebenfalls die Liste von 3-Tupeln (`move_count_list`) als Input. Die Funktion berechnet aus dieser Liste die durchschnittliche Prozentzahl an Zügen, welche von Stockfish oder der KI weniger benötigt werden. Diese werden für sowohl die KI(`avg_ki_better`), als auch für Stockfish zurückgegeben(`avg_stock_better`).

[ ]:
```
def get_average_difference(move_count_list):
    percentual_ki = []
    percentual_stock = []
    avg_ki_better = 0
    avg_stock_better = 0
    for ki_move, stock_move, diff in move_count_list:
        if diff < 0:
            percentual_ki.append(round(1 - (ki_move / stock_move), 4))
        elif diff > 0:
            percentual_stock.append(round(1 - (stock_move / ki_move),␣
↪4))
    if len(percentual_ki) != 0:
        avg_ki_better = sum(percentual_ki) / len(percentual_ki)
    if len(percentual_stock) != 0:
        avg_stock_better = sum(percentual_stock) / len(percentual_stock)
    return avg_ki_better, avg_stock_better
```

Die Ergebnisse des Vergleichs sollen abschließend in einer Datei gespeichert werden. Dies hat den Grund, dass bei einer großen Anzahl von S_n Mengen die Auswertung nicht in einem unübersichtlichen Konsolenfenster durchgeführt werden muss und man die Ergebnisse mehrfach betrachten kann. Hierfür wird die Funktion `write_result_to_file` definiert. Diese berechnet zunächst mit der `move_count_list` die Ergebnisse des Vergleichs. Hierzu werden die zuvor definierten Funktionen `compare_move_lists` und `get_average_difference` aufgerufen. Außerdem werden die prozentualen Ergebnisse, welche Engine/KI das Problem besser gelöst hat, innerhalb dieser Funktion berechnet. Die Ergebnisse werden letztendlich in die Datei mit dem Namen `filename`.txt in den Ordner `Tests` geschrieben. Zur Übersicht wird ein `sequence_index` mitgegeben werden, der eine Unterscheidung der Tests ermöglicht.

[ ]:
```
def write_result_to_file(filename, sequence_index, move_count_list):
```

```
    equal, ki_better, stockfish_better =␣
↪compare_move_lists(move_count_list)
    avg_ki_better, avg_stock_better =␣
↪get_average_difference(move_count_list)
    count = ki_better + stockfish_better + equal
    f = open("Tests/" + filename + ".txt", "a+")
    f.write("S_" + str(sequence_index) + ":\n")
    f.write("Stockfish war zu " + str(round((stockfish_better / count)␣
↪* 100, 2)) + "% besser.\n")
    f.write("Die KI war zu " + str(round((ki_better / count) * 100, 2))␣
↪+ "% besser.\n")
    f.write("Stockfish und die KI haben zu " + str(
        round((equal / count) * 100, 2)) + "% die gleichen Ergebnisse␣
↪erzielt.\n")
    f.write("Sofern die KI besser war, hat sie durchschnittlich " + str(
        round(avg_ki_better * 100, 2)) + "% weniger Züge benötigt.\n")
    f.write("Sofern Stockfish besser war, hat sie durchschnittlich " +␣
↪str(
        round(avg_stock_better * 100, 2)) + "% weniger Züge benötigt.
↪\n")
    f.close()
```

6.2.3 Implementierung der Testszenarien

In diesem Abschnitt werden die zuvor definierten Testszenarien implementiert. Diese erhalten eine Liste der S_n Mengen mit der verkürzten FEN Schreibweise (`s_n_sequence_short`) oder die Liste der S_n Mengen mit der vollständigen FEN Schreibweise (`s_n_sequence_fen`). Diese werden für die Berechnungen der Züge für die KI benötigt und müssen deshalb in mindestens einer Form für die Funktion vorliegen.

Das erste Testszenario sieht die Überprüfung eines Spielszenarios vor (`fen`). Dies geschieht in der Funktion `compare_fen_stockfish`. Das Ergebnis wird am Ende der Funktion in der Konsole ausgegeben.

[ ]:
```
def compare_fen_stockfish(fen, s_n_sequence_short):
    moves = calculate_all_moves(fen, s_n_sequence_short)
    stockfish_moves = stockfish_movelist(fen)
```

```
    if Moves is not None:
        print("AI needed " + str(len(moves)) + " moves to beat␣
↪Stockfish as Black.")
    else:
        print("AI found no way to beat Black.")

    print("Stockfish needed " + str(len(stockfish_moves)) + " moves to␣
↪win against itself.")
```

Das zweite Szenario sieht vor, eine beliebige Anzahl (`count`) zufälliger Boards zu vergleichen. Anhand der `count`-Variable werden aus den S_n Mengen zufällige Boards ausgewählt und der Vergleich zwischen der KI und Stockfish durchgeführt. Das Ergebnis wird mit `write_result_to_file` in einer Datei hinterlegt.

[ ]:
```
import random


def test_random_boards(count, s_n_sequence_fen, s_n_sequence_short):
    count_s_n = len(s_n_sequence_fen)
    move_count_list = []
    for board_c in range(count):
        rand_sequence = random.randint(0, count_s_n - 1)
        rand_board = random.randint(0,␣
↪len(s_n_sequence_fen[rand_sequence]) - 1)
        rand_fen = list(s_n_sequence_fen[rand_sequence])[rand_board]
        ki_moves = calculate_all_moves(rand_fen, s_n_sequence_short)
        stockfish_moves = stockfish_movelist(rand_fen)
        move_count = len(ki_moves)
        cmp_move_count = len(stockfish_moves)
        move_count_list.append(tuple((move_count, cmp_move_count,␣
↪move_count - cmp_move_count)))
        clear_output()
        print("Analyzed " + str(board_c + 1) + "/" + str(count))

    filename = "Random_" + str(count) + "_Compare_" + str(datetime.
↪today().replace(microsecond=0)).replace(":",

                                                                    ␣
↪                                    "_") + ".txt"
```

```
    write_result_to_file(filename, "random", move_count_list)
```

In diesem Beispiel werden 100 zufällige Spielsituationen miteinander verglichen:

[ ]:
```
test_random_boards(100, S_N_Sequence_fen, S_N_Sequence_fen_short)
```

Bei `compare_sequence_stockfish` wird für jedes Board einer S_n Menge überprüft, ob Stockfish oder die zuvor definierten Mengen die Situation effizienter lösen. In Form des `sequence_index` wird das n der S_n Menge übergeben. Zusätzlich kann ein Name für eine Datei angegeben werden, sodass mehrere Testergebnisse in einer Datei festgehalten werden können. Das Beschreiben der Datei erfolgt mit der Funktion `write_result_to_file`.

[ ]:
```
def compare_sequence_stockfish(sequence_index, s_n_sequence_fen,␣
↪s_n_sequence_short, g_filename=None):
    s_n = s_n_sequence_fen[sequence_index]
    move_count_list = []
    length = len(s_n)
    r = 0
    for board_fen in s_n:
        ai_moves = calculate_all_moves(board_fen, s_n_sequence_short)
        stockfish_moves = stockfish_movelist(board_fen)
        move_count = len(ai_moves)
        cmp_move_count = len(stockfish_moves)
        move_count_list.append(tuple((move_count, cmp_move_count,␣
↪move_count - cmp_move_count)))
        clear_output()
        print("Comparing S_" + str(sequence_index) + ":")
        print("Compared " + str(r + 1) + "/" + str(length))
        r += 1

    if g_filename is None:
        filename = "S_" + str(sequence_index) + "_Compare_" +␣
↪str(datetime.today().replace(microsecond=0)).replace(":",
                                                                    ␣
↪                                                  "_") + ".txt"
        write_result_to_file(filename, sequence_index, move_count_list)
    else:
```

```
        write_result_to_file(g_filename, sequence_index,␣
↪move_count_list)
```

Als Beispiel wird hier der Vergleich zwischen der KI und Stockfish für die Menge S_{20} vorgenommen:

[ ]:
```
compare_sequence_stockfish(20, S_N_Sequence_fen, S_N_Sequence_fen_short)
```

Das letzte Testszenario sieht vor, alle zuvor bestimmten Boards mit dem Lösungsweg von Stockfish zu vergleichen. Hierfür geht die Funktion `compare_all_sequences` alle S_n Mengen durch und vergleicht diese jeweils mit der Lösung von Stockfish. Alle Ergebnisse werden in eine Datei geschrieben.

[ ]:
```
def compare_all_sequences(s_n_sequence_fen, s_n_sequence_short):
    filename = "All_S_Compare_" + str(datetime.today().
↪replace(microsecond=0)).replace(":", "_") + ".txt"
    for i in range(len(s_n_sequence_fen)):
        print("Comparing S_" + str(i) + "...")
        compare_sequence_stockfish(i, s_n_sequence_fen,␣
↪s_n_sequence_short, filename)
```

[ ]:
```
compare_all_sequences(S_N_Sequence_fen, S_N_Sequence_fen_short)
```

Für die visuelle Darstellung der Züge wird die Funktion `show_movelist` definiert. Diese zeigt auf einem `chess.Board` Objekt eine mitgegebene Liste von Zügen zeitverzögert an. Für die Darstellung erhält die Funktion die `fen`, die die Spielsituation zu Beginn der Partie darstellt. Weiter muss eine Liste von Spielzügen (`moves`), die dargestellt werden sollen, der Funktion mitgegeben werden.

[ ]:
```
def show_movelist(fen, moves):
    presentation_board = chess.Board(fen)
    display(presentation_board)
    time.sleep(2)
    for move in moves:
        presentation_board.push(move)
        clear_output(wait=True)
        display(presentation_board)
        time.sleep(2)
```

7 Überprüfen der S_n Mengen

Bei der Bestimmung der S_n Mengen wurden alle möglichen Züge rückwärts durchgeführt. Damit die Retrograde Analyse erfolgreich zum Gewinnen von Endspielsituationen angewandt werden kann, muss diese Berechnung erfolgreich gewesen sein. Um sicherzustellen, dass die Situationen, die z. B. in S_{10} zu finden sind, auch wirklich in ≤ 10 Zügen beendet werden, wurden in diesem Notebook Funktionen definiert, die diesen Aspekt überprüfen. Hierfür werden zuerst die berechneten S_n Mengen aus der Datei geladen.

[ ]:
```
S_N_Sequence_fen, S_N_Sequence_fen_short = load_s_n_fens("S_n_seq_rook")
```

Die erste Funktion zur Überprüfung, ist `fen_in_lower_sequence`. Diese überprüft für eine mitgegebene `fen`, ob diese in einer S_n Menge liegt, für die gilt:

$$n < sequence\_index$$

Die Funktion gibt hierfür einen booleschen Wert zurück.

[ ]:
```
def fen_in_lower_sequence(fen, sequence_index, s_n_short):
    for s in s_n_short[:sequence_index]:
        if fen in s:
            return True
    return False
```

Eine weitere Überprüfung liegt darin, dass keine der Schachbretter in eine zu niedrige S_n Menge eingeordnet wurde. Dies wird mit der Funktion `every_move_of_sequence_in_lower` umgesetzt. Hierfür erhält sie eine `sequence_index`, die überprüft werden soll. Für jede Spielsituation wird Folgendes überprüft:

$$\forall board \in S_n : move \in board.legal\_moves \implies board.push(move) \in S_m \wedge m < n$$

Als Ergebnis gibt die Funktion eine boolesche Variable zurück, die `True` zurückgibt, falls die Bedingung erfüllt ist, ansonsten `False`.

[ ]:
```
def every_move_of_sequence_in_lower(sequence_index, s_n_fen, s_n_short):
    print("Checking S_" + str(sequence_index) + "...")
```

```
    sequence_fen = s_n_fen[sequence_index]
    for fen in sequence_fen:
        cur_board = chess.Board(fen)
        if not cur_board.turn:
            for move in cur_board.legal_moves:
                cur_board.push(move)
                cur_short = get_board_and_turn(cur_board.fen())
                if not fen_in_lower_sequence(cur_short, sequence_index,␣
↪s_n_short):
                    print("Fen: " + cur_short + " not in lower S")
                    return False
                cur_board.pop()
            return True
        else:
            in_lower = False
            for move in cur_board.legal_moves:
                cur_board.push(move)
                cur_short = get_board_and_turn(cur_board.fen())
                if fen_in_lower_sequence(cur_short, sequence_index,␣
↪s_n_short):
                    in_lower = True
                cur_board.pop()
            return in_lower
```

Die folgende Zelle führt diesen Test durch.

[ ]:
```
for i in range(len(S_N_Sequence_fen)):
    every_move_of_sequence_in_lower(i, S_N_Sequence_fen,␣
↪S_N_Sequence_fen_short)
```

Die Funktion `find_move_count` kann für folgende Überprüfung verwendet werden:

$$\forall board \in S_n \implies max(find\_move\_count(board)) = n$$

Hierbei berechnet `find_move_count` jeden Ablauf einer Schach-Partie, bis ein Schachmatt erzielt worden ist. Da aber die Spielsituation in die Menge S_n zugeordnet wurde, muss auf jeden Fall eine dieser Zahlen den Wert n betragen, da ansonsten das Schachbrett der falschen S_n Menge zugeordnet wurde.

Die Implementierung der Funktion `find_move_count` erfolgt rekursiv. Die Funktion ruft sich selbst auf, bis ein Schachmatt erzielt worden ist. Für einen derartigen Aufruf wird die `sequence_index`, in der sich die `fen` befindet, mitgegeben. Außerdem muss das `move_set`, dass auch als Rückgabewert fungiert, mitgegeben werden, da darin die Anzahl der Züge gespeichert werden, die zum Lösen jeglicher Spielabläufe benötigt werden. Der Parameter `move_count` resultiert ebenfalls durch die Rekursion, da in ihr die aktuelle Anzahl an ausgeführten Spielzügen gespeichert wird. Aufgrund des Aufrufs der Funktion `find_situation_in_sequence` wird zusätzlich noch die Liste der gekürzten S_n Mengen beigefügt (`s_n_short`). Die Funktion gibt letztendlich eine Menge an Spielzügen zurück, die für die unterschiedlichen Abläufe der Spielsituation benötigt wurden.

[ ]:
```
def find_move_count(sequence_index, fen, move_count, move_set,␣
↪s_n_short):
    cur_board = chess.Board(fen)
    moves = cur_board.legal_moves
    if moves.count() == 0 and sequence_index == 0:
        move_set.add(move_count)
        return move_set
    if not cur_board.turn:
        # print("Black:")
        for move in cur_board.legal_moves:
            cur_board.push(move)
            move_count += 1
            cur_fen = cur_board.fen()
            new_seq = find_situation_in_sequence(cur_fen, s_n_short)
            # print(new_seq, ";" , cur_fen)
            move_set = find_move_count(new_seq, cur_fen, move_count,␣
↪move_set, s_n_short)
            move_count -= 1
            cur_board.pop()
    else:
        # print("White:")
        for move in cur_board.legal_moves:
            cur_board.push(move)
            cur_fen = get_board_and_turn(cur_board.fen())
            if fen_in_lower_sequence(cur_fen, sequence_index,␣
↪s_n_short):
                new_seq = find_situation_in_sequence(cur_fen, s_n_short)
```

```
                move_count += 1
                move_set = find_move_count(new_seq, cur_fen,␣
↪move_count, move_set, s_n_short)
                move_count -= 1
            cur_board.pop()
    return move_set
```

[ ]:
```
find_move_count(20, "8/8/8/3K4/8/5R2/2k5/8 b - - 0 1", 0, set(),␣
↪S_N_Sequence_fen_short)
```

Zur Überprüfung, dass nun Board der falschen S_n Menge zugeordnet wurde, wird die Funktion `check_boards_in_correct_sequence` definiert. Diese überprüft für einen `sequence_index` in einer mitgegebenen Liste von S_n Mengen (`s_n_fen`), ob die Spielsituationen der Menge an dem Index `sequence_index` auch maximal n Züge bis zum Ende der Partie benötigen. Dies gelingt mit der Funktion `find_move_count`, die zusätzlich noch die gekürzte Schreibweise der S_n Mengen benötigt (`s_n_short`). Die Funktion gibt in Form einer Konsolenausgabe an, welche der FENs in eine falsche S_n Menge zugeordnet worden sind.

[ ]:
```
def check_boards_in_correct_sequence(sequence_index, s_n_fen,␣
↪s_n_short):
    invalid = 0
    print("Comparing S_" + str(sequence_index) + "...")
    for fen in s_n_fen[sequence_index]:
        move_set = find_move_count(sequence_index, fen, 0, set(),␣
↪s_n_short)
        if max(move_set) != sequence_index:
            invalid += 1
            print(fen)
    print("Invalid boards: " + str(invalid))
```

[ ]:
```
check_boards_in_correct_sequence(4, S_N_Sequence_fen,␣
↪S_N_Sequence_fen_short)
```

Zusätzlich zum rechnerischen Überprüfen der Ergebnisse kann eine andere Methode zum Verifizieren verwendet werden. Schach-Endspieldatenbanken sind keine neue Erfindung und wurden bereits von anderen Forschern entwickelt. Ein lang anhaltendes Projekt, welches gegenwärtig Datenbanken für Situationen mit bis zu 7 Figuren auf dem Spielfeld anbietet ist Syzygy. Das Projekt hält zwei Datentypen für alle Spiel-

situationen vor: WDL Daten und DTZ Daten.

WDL steht für Win / Draw / Loss und gibt dem Nutzer eine Information über den Wert einer Spielsituation. Eine Anfrage an die Datenbank mit einer Situation wird mit einem der folgenden Werte beantwortet: -2, -1, 0, 1, 2. Positive Werte implizieren, dass bei perfektem Spiel der aktuelle Spieler gewinnt, negative Werte bedeuten, dass der aktuelle Spieler verliert. Die 0 bedeutet, dass das Spiel (wenn beide Seiten perfekt spielen) in einem Unentschieden endet. Eine Zwei ist ein sicherer Sieg / Verlust, während eine Eins in einem Gewinn oder unentschieden mittels der 50-Zug Regel enden kann.

Die Interessantere der Dateien ist die DTZ-Datei. DTZ steht für Distance to Zero. Die DTZ Tabelle enthält Werte von -100 bis 100. Positiv, Negativ und Null kann genau wie WDL interpretiert werden. Die Zahlen von -100 bis -1 und 1 bis 100 geben die Anzahl der Halbzüge bis zu einem Gewinn (oder Reset der 50-Züge Regel) an. Stetiges verringern einer positiven DTZ führt also zu einem Gewinn.

Die DTZ-Zahl einer Spielsituation kann mit dem n verglichen werden, in welcher Menge S_n diese Situation in der `.chessAI` eingeordnet wurde. Stimmen diese Zahlen überein, war die Berechnung korrekt.

Hinweis: Je nach Ausführung der Syzygy-Tabellen werden halbe oder ganze Züge gespeichert, es muss daher beim Vergleich eine Toleranz von einem n akzeptiert werden.

Die Funktion `compare_with_syzygy()` führt diesen Vergleich durch und gibt die Anzahl falsch eingeordneter Situationen zurück.

[ ]:
```
def compare_with_syzygy(syzygy, s_n_sequence):
    counter = 0

    for n in range(len(s_n_sequence)):
        count = 0
        for fen in s_n_sequence[n]:
            chess_board = chess.Board(fen)
            if n != abs(syzygy.probe_dtz(chess_board)) != n + 1:
                count += 1
        print(f"S{n}: Syzygy believes {count} of {len(s_n_sequence[n])}␣
↪Situations are wrongly placed in the sequence.")
    print(f"Syzygy believes {counter} Situations are wrongly placed in␣
↪the sequence.")
    return count
```

Die Syzygy-Dateien müssen sich entweder im Ordner `./syzygy` befinden oder der

Pfad angepasst werden.

[ ]:
```
SYZYGY = chess.syzygy.Tablebase()
SYZYGY.add_directory("./syzygy")
```

[ ]:
```
compare_with_syzygy(SYZYGY, S_N_Sequence_fen)
```

Literaturverzeichnis

Carlstedt, J.: Die kleine Schachschule, Humboldt Verlag, o.O. 2014

chess.com o.V.: "Chess Engine", o. D., https://www.chess.com/terms/chess-engine, Einsichtnahme: 13.04.2022

Chess Programming Wiki o.V.: "Flipping Mirroring and Rotating ", o. D., https://www.chessprogramming.org/Flipping_Mirroring_and_Rotating, Einsichtnahme: 13.04.2022

Fiekas, Niklas: SSyzygy endgame tablebases", o.D., https://syzygy-tables.info/, Einsichtnahme: 13.04.2022