

Einleitung

TODO

1. Frontend

Das Frontend wird mit Vue.js entwickelt und soll sich durch ein verspieltes, dennoch übersichtliches und intuitives Design auszeichnen. Die Interaktion mit dem Schachbrett erfolgt über die Chessground-Bibliothek und die mit dieser bereitgestellten BoardAPI, die eine effiziente und flexible Brettmanipulation sowie dessen Visualisierung ermöglicht.

Beim Styling habe ich mich für den BEM-Syntax entschieden, um ein besseres Verständnis für CSS-Prinzipien zu erlangen und gleichzeitig eine professionelle Benennung meiner CSS Elemente zu garantieren.

2. Backend

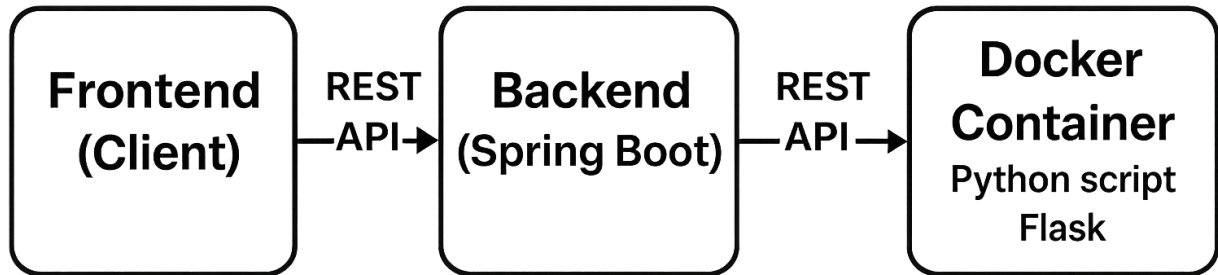
Das Backend wird als REST-API umgesetzt und übernimmt die Evaluierung und Berechnung von Zügen. Es wird in einem Docker-Container mit Flask betrieben, um eine isolierte, skalierbare und leicht wartbare Architektur zu gewährleisten. Python wurde als Programmiersprache für die Evaluierungsfunktion gewählt, da es weit verbreitet in KI-Anwendungen und Algorithmik ist und ich zudem meine Python Fähigkeiten ausbauen möchte.

2.1 REST-API

Die REST-API dient als Schnittstelle zwischen dem Client und dem Backend. Der erste implementierte Endpunkt ist eine POST-Request, die den besten Zug für eine gegebene Brettstellung zurück liefert.

Dabei dient das Backend (Spring Boot) als Vermittler zwischen Client und unserem Docker Container, welcher das Python Skript beinhaltet. Dort werden Züge evaluiert sowie oder MINIMAX-

Algorithmus ausgeführt.



2.2 Docker

Um eine modulare und leicht wartbare Architektur sicherzustellen, wird das Python Programm, wie bereits erwähnt, in einem separaten Docker-Container betrieben. Die grundlegende Struktur des Python-Services umfasst folgende Komponenten:

2.2.1 Verzeichnisstruktur:

- `/python-service` enthält alle notwendigen Dateien für den Python-Dienst, darunter:
- `requirements.txt` zur Definition der benötigten Python-Abhängigkeiten
- `validation.py` für die Implementierung der Validierungsfunktion
- `constants.py` zur Speicherung der Konstanten für die Evaluierungsfunktion wie PSQT oder Zugmuster der Figuren
- `Dockerfile` zur Containerisierung des Backends
- `evaluations-tests` für die Unittest dieses Projekts
- `searchfunction` enthält die eigentliche Suchfunktion

2.2.2 Flask als Webframework:

Die neueste Version von Flask wird verwendet, um eine effiziente und leichtgewichtige REST-API bereitzustellen. Diese wird genutzt um zwischen SpringBoot und dem Python-Service zu kommunizieren.

2.2.3 Docker-Container:

Zunächst habe ich ein schlankes Docker-Image (`python:3.10-slim`) gewählt, um die Angriffsfläche zu minimieren und eine ressourcenschonende Umgebung zu gewährleisten. Dies führt zu einer besseren Performance und reduzierten Abhängigkeiten. Da es hier jedoch Probleme mit dem Dockercontainer gab, habe ich mich doch für `python:3.10` entschieden, die zuvor beschriebenen Probleme wurden somit beseitigt.

Um dafür zu sorgen, dass sich der Dockercontainer selbstständig aktualisiert, Änderungen also selbstständig übernommen werden, ohne den Container mit `docker-compose up` neu bauen zu müssen, habe ich mich hier dafür entschieden diese Funktion mithilfe von Volumes zu ermöglichen.

Durch die Erweiterung der `docker-compose.yml` um

```
1 volumes:
2   - ./python-service:/app
```

erkennt der Dockercontainer nun, falls Änderungen innerhalb des `python-service` Verzeichnis vorgenommen werden und startet den Container automatisch neu.

2.3 Unittests

Da viele Grundlegenden Funktionen bereits zu Beginn des Projektes geschrieben wurden und über die Zeit sicherlich verbessert, erweitert oder überarbeitet werden, sind ausgiebige Tests an dieser Stelle sehr wichtig.

Mit diesen können zu jeder Zeit die Kernfunktionen der Engine überprüft werden und sichergestellt werden, dass getätigte Änderungen keinen Einfluss auf die Funktionsweise haben.

Mit `python -m unittest evaluation-tests -v` können die Tests ausgeführt werden. Für mehr Informationen kann auch `coverage` genutzt werden.

- `coverage run -m unittest evaluation-tests` : Führt Tests aus und erstellt einen Coverage-Report
- `coverage report` : Zeigt Coverage-Report in CLI an.
- `coverage html` : Erstellt HTML Ansicht der Coverage.

3. Validierungsfunktion

3.1 Board Representation

Zunächst müssen wir eine geeignete Repräsentation des Schachbrettes wählen um die Figuren bestimmten Feldern zuzuordnen, dessen Werte zu evaluieren und Züge zu berechnen.

Anfangs wollte ich das Spielfeld als 1D Array übersetzen. Da Speicherplatz in diesem Projekt jedoch keine limitierende Ressource ist, habe ich mich an der Stelle doch für einen 2D Array entschieden, da somit viele Berechnungen und Zugriffe vereinfacht werden.

Es muss also der `FEN` , welcher vom Frontend via Post-Request übergeben wird in einen 8×8 2D Array übersetzt werden.

Ein vollständiger FEN-String besteht aus **6 durch Leerzeichen getrennten Feldern**:

1. Stellung der Figuren

- Beschreibt die Belegung der 8 Reihen (von oben nach unten)
- Kleinbuchstaben = schwarze Figuren, Großbuchstaben = weiße Figuren
- Zahlen = Anzahl leerer Felder

2. Zugrecht

- **w** → Weiß ist am Zug
- **b** → Schwarz ist am Zug

3. Rochaderechte

- **K** → Weiß kann kurz rochieren
- **Q** → Weiß kann lang rochieren
- **k** → Schwarz kann kurz rochieren
- **q** → Schwarz kann lang rochieren
- **-** → Keine Rochaderechte mehr vorhanden

4. En-passant-Ziel

- Gibt das Feld an, auf das en passant geschlagen werden kann
- **-** → Kein En-passant möglich

5. Halbzugzähler

- Anzahl der Halbzüge seit dem letzten Bauernzug oder Schlagzug
- Wird für die 50-Züge-Regel verwendet

6. Vollzugzähler

- Startet bei 1 und wird nach jedem Zug von Schwarz erhöht

Die Funktion `fen_to_array` übernimmt genau diese Funktion und erstellt aus der vom Frontend übergebenen FEN einen 2D Array.

```
1  # Convert FEN into 2D Array
2  def fen_to_array(fen):
3
4      # 2D Array 8x8
5      chessboard = [[0 for _ in range(8)] for _ in range(8)]
6
7      current_field = 0
8      current_row = 0
9      current_column = 0
10
11     for char in fen:
12
13         if char.isalpha():
14             current_row = current_field // 8
15             current_column = current_field % 8
16             chessboard[current_row][current_column] = char
17             current_field += 1
18
19         elif char.isdigit():
20             current_field += int(char)
21
22         elif char == " ":
```

```

23         break
24
25     return chessboard

```

Mithilfe dieser Darstellung können wir nun beginnen unseren aktuellen Schachbrett-Zustand zu evaluieren und diesem einen Wert zuzuordnen.

3.2 Piece Counting

Um die Basis unserer Evaluierungsfunktion zu setzen, müssen wir zunächst den Figuren verschiedene Werte zuweisen, eine gängige Bewertung ist die folgende:

```

1  class piece(IntEnum):
2      QUEEN = 900
3      ROOK = 400
4      BISHOP = 300
5      KNIGHT = 300
6      PAWN = 100
7      KING = 0

```

Diese hat sich bereits in anderen Modellen gut behauptet und kann somit guten Gewissens übernommen werden.

Es gibt auch Varianten in denen der Bishop mehr gewertet wird als der Knight, etwa mit 350, wir bleiben jedoch bei der ersten Variante aus gegebenen Gründen.

3.3 Piece Square Tables

Die für dieses Projekt gewählte PSQT sind von [Stockfish](#), somit haben wir garantiert eine gute Basis um uns auf den Kern dieses Projektes, der Evaluierungsfunktion und dem Suchalgorithmus zu beschäftigen.

Zunächst habe ich einen PSQT für jede Figur für den Spielbeginn festgelegt:

```

1  class Psqt:
2
3      PAWN_PSQT = [
4          0, 0, 0, 0, 0, 0, 0, 0,
5          98, 134, 61, 95, 68, 126, 34, -11,
6          -6, 7, 26, 31, 65, 56, 25, -20,
7          -14, 13, 6, 21, 23, 12, 17, -23,
8          -27, -2, -5, 12, 17, 6, 10, -25,
9          -26, -4, -4, -2, 4, 4, -4, -24,
10         -35, -17, -13, -10, -11, -13, -17, -37,
11         0, 0, 0, 0, 0, 0, 0, 0
12     ]
13
14     KNIGHT_PSQT = [
15         -167, -89, -34, -49, 61, -97, -15, -107,
16         -73, -41, 72, 36, 23, 62, 7, -17,

```

```

17     -47, 60, 37, 65, 84, 129, 73, 44,
18     -9, 17, 19, 53, 37, 69, 18, 22,
19     -13, 4, 16, 13, 28, 19, 21, -8,
20     -23, -9, 12, 10, 19, 17, 25, -16,
21     -29, -53, -12, -3, -1, 18, -14, -19,
22     -105, -21, -58, -33, -17, -28, -19, -23
23 ]
24
25 BISHOP_PSQT = [
26     -29, 4, -82, -37, -25, -42, 7, -8,
27     -26, 16, -18, -13, 30, 59, 18, -47,
28     -16, 37, 43, 40, 35, 50, 37, -2,
29     -4, 5, 19, 50, 37, 37, 7, -2,
30     2, -8, 16, 29, 27, 41, 12, 9,
31     -6, 3, 9, 23, 24, 35, 6, 10,
32     -6, 0, 4, -7, -1, -5, 24, -18,
33     -17, -17, -14, -8, -6, -10, -15, -21
34 ]
35
36 ROOK_PSQT = [
37     32, 42, 32, 51, 63, 9, 31, 43,
38     27, 32, 58, 62, 80, 67, 26, 44,
39     -5, 19, 26, 36, 17, 45, 61, 16,
40     -24, -4, 4, 1, -7, -6, -9, -16,
41     -36, -27, -20, -12, -1, -7, 3, -37,
42     -39, -18, -9, 3, 6, -6, -3, -45,
43     -44, -16, -20, -9, -1, 11, -6, -71,
44     -19, -13, 1, 17, 16, 7, -37, -26
45 ]
46
47 QUEEN_PSQT = [
48     -28, 0, 29, 12, 59, 44, 43, 45,
49     -24, -39, -5, 1, -16, 57, 28, 54,
50     -13, -17, 7, 8, 29, 56, 47, 57,
51     -27, -27, -16, -16, -1, 17, -2, 1,
52     -9, -26, -9, -10, -2, -4, 3, -3,
53     -14, 2, -11, -2, -5, 2, 14, 5,
54     -35, -8, 11, 2, 8, 15, -3, 1,
55     0, 0, 0, 0, 0, 0, 0, 0
56 ]
57
58 KING_PSQT = [
59     -65, 23, 16, -15, -56, -34, 2, 13,
60     29, -1, -20, -7, -8, -4, -38, -29,
61     -9, 24, 2, -16, -20, 6, 22, -22,
62     -17, -20, -12, -27, -30, -25, -14, -36,
63     -49, -1, -27, -39, -46, -44, -33, -51,
64     -14, -14, -22, -46, -44, -30, -15, -27,
65     1, 7, -8, -64, -43, -16, 9, 8,
66     -15, 36, 12, -54, 8, -28, 24, 14
67 ]

```

Diese PSQT können später noch, wie in der `theory.md` beschrieben erweitert werden. Für eine einfache Evaluierungsfunktion reichen uns diese jedoch völlig aus.

Um nicht für jede Figur (Schwarz und Weiß) verschiedene PSQT anlegen zu müssen, berechnen wir die Werte wie folgt:

```
1  # Get the position value from the piece-square table for a given piece and
   position
2
3  def get_psqt_value(field, piece, is_white):
4
5
6
7      psqt = get_psqt_table(piece, is_white) # Get the appropriate piece-square
   table
8
9      kingOrQueen = piece.upper() == "K" or piece.upper() == "Q"
10
11     # For black pieces or kings/queens, use the mirrored square value
12
13     if not is_white or kingOrQueen:
14
15         return psqt[63 - field] # Return mirrored value
16
17     else:
18
19         return psqt[field] # Return normal value
```

Die Funktion `calc_piece_value_with_psqt(chessboard)`, welche `get_psqt_value(field, piece, is_white)` aufruft, durchläuft Feld für Feld unser Chessboard (2D Array) und berechnet den Wert für das aktuell betrachtete Feld und die sich darauf befindende Figur. Für schwarze Figuren reicht es somit also, wenn wir `psqt[field]` zurückzugeben. Um den korrekten Wert für die weißen Figuren zurückzugeben, müssen wir den gespiegelten Wert zurückgeben `psqt[63 - field]`.

Der einzige Fall in dem dieses Vorgehen nicht funktioniert ist bei der Dame oder dem König, da diese in diesem Sinne vertauscht sind. Somit habe ich für diese beiden Figuren einzelne PSQT angelegt, welche Horizontal gespiegelt sind.

`get_psqt_table(piece, is_white)` sorgt dafür, dass auch in diesem Fall der richtige PSQT zurückgegeben wird.

```
1  ...
2
3  QUEEN_PSQT_BLACK = [
4      45, 43, 44, 59, 12, 29, 0, -28,
5      54, 28, 57, -16, 1, -5, -39, -24,
6      57, 47, 56, 29, 8, 7, -17, -13,
7      1, -2, 17, -1, -16, -16, -27, -27,
8      -3, 3, -4, -2, -10, -9, -26, -9,
9      5, 14, 2, -5, -2, -11, 2, -14,
10     1, -3, 15, 8, 2, 11, -8, -35,
11     0, 0, 0, 0, 0, 0, 0, 0
12 ]
13
14 KING_PSQT_BLACK = [
```

```

15     13, 2, -34, -56, -15, 16, 23, -65,
16     -29, -38, -4, -8, -7, -20, -1, 29,
17     -22, 22, 6, -20, -16, 2, 24, -9,
18     -36, -14, -25, -30, -27, -12, -20, -17,
19     -51, -33, -44, -46, -39, -27, -1, -49,
20     -27, -15, -30, -44, -46, -22, -14, -14,
21     8, 9, -16, -43, -64, -8, 7, 1,
22     14, 24, -28, 8, -54, 12, 36, -15
23 ]
24
25 ...

```

TODO erweitern um Middlegame und Endgame

4. Suchalgorithmus

Nachdem wir nun eine Grundlegende Bewertung des Spielzustands vornehmen können, können wir nun auch unseren generischen MiniMax-Algorithmus Implementieren.

4.1 Generischer MinMax Algorithmus

Betrachten wir zunächst einen Pseudocode, welcher den generischen MiniMax Algorithmus beschreibt.

```

1  def MINIMAX(position, depth, isMaximizingPlayer):
2      if depth == 0 or game_over(position):
3          return evaluate(position) # Bewertung der Stellung
4
5      if isMaximizingPlayer:
6          bestValue = -∞
7          for each child in generate_moves(position):
8              value = MINIMAX(child, depth - 1, false)
9              bestValue = max(bestValue, value)
10         return bestValue
11
12     else:
13         bestValue = ∞
14         for each child in generate_moves(position):
15             value = MINIMAX(child, depth - 1, true)
16             bestValue = min(bestValue, value)
17         return bestValue

```

Wir sehen hier, dass noch keine Verbesserungen wie Alpha-Beta Pruning vorgenommen wurden.

Als erstes kümmern wir uns um die beiden Methoden:

- `game_over(position)` -> berechnet ob Position im Schachmatt resultiert.
- `generate_moves(position)` -> berechnet alle möglichen Züge einer Seite basierend auf `position`

4.1.1 generate_moves(position)

Genau an dieser Stelle habe ich mich auch dazu entschieden, wie oben bereits angesprochen, die Darstellung des Schachbrettes in einen 2D Array zu ändern und nicht mithilfe eines 1D Arrays die Züge zu berechnen, da somit der Zugriff auf Zeilen und Spalten deutlich einfacher fällt und auch deutlich intuitiver ist. Der Speicherverbrauch ist zudem in diesem Projekt keine limitierende Ressource.

In professionelle Schachengines werden oft Bitmaps verwendet

- Professionell wären Bitmaps -> Später vielleicht

○ **TODO:** Kurz erklären

Mit unserem Chessboard in 2D-Darstellung, können wir einfach über alle Felder iterieren.

Dabei erstellen wir ein Dictionary, welches jeder Figur (row, column) eine endliche Menge an möglichen Zügen [(row1, column1), (row2, column2), ...] zuweist.

```
1  def generate_moves(chessboard, is_white):
2
3      moves = {} # Dic for all moves
4
5      for i in range(8):
6          for j in range(8):
7
8              piece = chessboard[i][j]
9
10             if piece == 0:
11                 continue
12
13             if piece.isupper() and is_white:
14                 match piece:
15                     case 'P': moves.setdefault((i, j), []).extend(pawn_moves(i, j, True,
chessboard))
16                     case 'B': moves.setdefault((i, j), []).extend(bishop_moves(i, j, True,
chessboard))
17                     case 'N': moves.setdefault((i, j), []).extend(knight_moves(i, j, True,
chessboard))
18                     case 'R': moves.setdefault((i, j), []).extend(rook_moves(i, j, True,
chessboard))
19                     case 'Q': moves.setdefault((i, j), []).extend(queen_moves(i, j, True,
chessboard))
20                     case 'K': moves.setdefault((i, j), []).extend(king_moves(i, j, True,
chessboard))
21
22             elif piece.islower() and not is_white:
23
24                 match piece:
25                     case 'p': moves.setdefault((i, j), []).extend(pawn_moves(i, j, False,
chessboard))
26                     case 'b': moves.setdefault((i, j), []).extend(bishop_moves(i, j, False,
chessboard))
```

```

27         case 'n': moves.setdefault((i, j), []).extend(knight_moves(i, j, False,
chessboard))
28         case 'r': moves.setdefault((i, j), []).extend(rook_moves(i, j, False,
chessboard))
29         case 'q': moves.setdefault((i, j), []).extend(queen_moves(i, j, False,
chessboard))
30         case 'k': moves.setdefault((i, j), []).extend(king_moves(i, j, False,
chessboard))
31
32     return moves

```

Um die Berechnung der Züge der einzelnen Figuren zu vereinfachen habe ich zudem die Zugmuster in `constants.py` gespeichert

```

1     class Piece_moves: # Pawn is handled seperatly
2
3         Knight = [(-2, -1), (-2, 1), (2, -1), (2, 1), (-1, -2), (-1, 2), (1, -2), (1,
2)] # Springer
4         Bishop = [(-1, -1), (-1, 1), (1, -1), (1, 1)] # Läufer (diagonal)
5         Rook = [(0, -1), (0, 1), (-1, 0), (1, 0)] # Turm (gerade)
6         King = [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]
# König (wie Dame, aber nur 1 Feld)
7         Queen = [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]
# Queen

```

Dabei ist wichtig zu betrachten, dass Pawns einzeln gehandhabt werden.

Obwohl Pawns zunächst wie die "einfachste" Figur wirken, sind diese tatsächlich in der Berechnung ihrer Züge etwas anders als der Rest der Figuren.

Sie können ein Feld nach vorne, von ihrem Startfeld aus aber auch zwei Felder nach vorne springen.

Außerdem schlagen Pawns anderes als die anderen Figuren nicht in ihrem normalen Zugmuster, sondern können nur nach links-vorne oder recht-vorne schlagen, was sie in der Art und Weise der Berechnung besonders macht.

Hier die Funktion zur Berechnung der Bauernzüge:

```

1     def pawn_moves(field_row, field_column, is_white, chessboard):
2
3         possible_moves = []
4
5         if is_white:
6             forward_row = field_row - 1
7             start_row = 6
8             direction = -1 # Weiß zieht nach oben
9         else:
10            forward_row = field_row + 1
11            start_row = 1
12            direction = 1 # Schwarz zieht nach unten
13
14            # Ein Feld vorwärts, wenn frei

```

```

15     if chessboard[forward_row][field_column] == 0:
16         possible_moves.append((forward_row, field_column))
17
18         # Zwei Felder vorwärts, wenn Startposition und beide Felder frei
19         if field_row == start_row and chessboard[forward_row + direction]
[forward_row + direction][field_column] == 0:
20             possible_moves.append((forward_row + direction, field_column))
21
22         # Schlagzüge nach links und rechts prüfen
23         for side in [-1, 1]: # -1 = links, +1 = rechts
24             new_column = field_column + side
25             if 0 <= new_column <= 7: # Prüfen, ob innerhalb der Spielfeldgrenzen
26                 target_piece = chessboard[forward_row][new_column]
27                 if target_piece != 0 and is_enemy(is_white, target_piece): # Feindliche
Figur dort?
28                     possible_moves.append((forward_row, new_column))
29
30     return possible_moves
31

```

On Passant Möglichkeiten werden hier noch nicht in Betracht gezogen, genauso wie Rochaderechte bei der Berechnung der Königzüge. Ich wollte zunächst mal eine einfache Version meines Algorithmus zum laufen bekommen, danach wird die Engine um genannte Funktionen erweitert.

Die Berechnung der Züge der restlichen Figuren hat im Prinzip immer den selben Aufbau, als Beispiel bietet sich die Queen besonders gut an, da diese die Zugmuster des Rooks, Bishops und Königs vereint:

```

1  def queen_moves(field_row, field_column, is_white, chessboard):
2
3      move_pattern = constants.Piece_moves.Queen
4      possible_moves = []
5
6      for direction in move_pattern:
7          row, column = field_row, field_column # Startposition
8
9          while in_bound(row + direction[0], column + direction[1]):
10
11              row += direction[0] # direction[0] stores vertical movement
12              column += direction[1] # direction[1] stores horizontal movement
13
14              if chessboard[row][column] == 0:
15                  possible_moves.append((row, column))
16
17              elif chessboard[row][column] != 0 and is_enemy(is_white, chessboard[row]
[column]):
18                  possible_moves.append((row, column))
19                  break # Gegner blockiert weitere Bewegung
20
21              else: # Eigene Figur blockiert Bewegung
22                  break
23
24      return possible_moves

```

Die restlichen Zugberechnungen können der `evaluation.py` entnommen werden.

Wir können nun also bereits alle möglichen Züge einer Seite (`is_white`) berechnen, ausgenommen von On Passant und Rochade

4.2 Matt erkennen

Die Matt-Erkennung funktioniert nach dem folgenden Schema:

- Berechne alle Züge für eine Seite
- Betrachte anschließend ob König angegriffen wird
 - Ja -> Zug nicht legal
 - Nein -> Zug legal

```
1  def is_check(chessboard, is_white):
2
3      all_opponent_moves = generate_moves(chessboard, not is_white) # Returns all
    possible moves of opponent
4      king_field = get_king_field(chessboard, is_white)
5
6      for key, value in all_opponent_moves.items(): # Iterate through all opponents
    moves
7          if (len(value) != 0):
8              for move in value:
9
10                 if move == king_field: # King is attacked -> check
11                     return True
12     return False # King is not attacked
```

Noch genauer können wir mit der Funktion `game_over` zurückgeben, ob es sich um kein Matt, Matt oder Patt handelt.

Dazu berechnen wir einfach die legalen Züge und wissen somit falls...

- ...keine Legalen Züge und Schach -> Matt
- ...keine Legalen Züge und kein Schach -> Patt

```
1  def game_over(chessboard, is_white):
2
3      legal_moves = len(generate_legal_moves(chessboard, is_white))
4
5      if legal_moves != 0:
6          return False, None
7      elif is_check:
8          return True, 0
9      else:
10         return True, 1
```

Da wir nun sowohl `game_over` berechnen können, sowie alle legalen Züge einer Seite mit `generate_moves`, können wir nun eine erste, generische Version unseres Suchalgorithmus implementieren:

```
1  # Minimax algorithm implementation to find the best move
2  def MINIMAX(chessboard_object, depth, is_white, move):
3      global counter
4      counter += 1
5      logger.debug(counter)
6
7      chessboard = chessboard_object.chessboard
8
9      logger.debug(f"MINMAX SEARCH: {counter}")
10     logger.debug(chessboard)
11
12     # Base case: if the maximum depth is reached
13     if depth == 0:
14         # Check if the game is over (checkmate or stalemate)
15         game_over, reason = validation.game_over(chessboard_object, is_white)
16
17         # If the game is not over, evaluate the current position
18         if not game_over: # not Checkmate or Stalemate
19             return Move_with_value(move.start, move.end,
validation.evaluate_position(chessboard)) # Evaluate the position
20
21         # If the game is over and it's the maximizer's turn (white)
22         if is_white: # Checkmate or Stalemate and Maximizer
23             return Move_with_value(move.start, move.end, INFINITY) if reason == 0 else
Move_with_value(move.start, move.end, 0) # Return infinity for checkmate, 0 for
stalemate
24
25         # If the game is over and it's the minimizer's turn (black)
26         else: # Checkmate or Stalemate and Minimizer
27             return Move_with_value(move.start, move.end, NEG_INFINITY) if reason == 0
else Move_with_value(move.start, move.end, 0) # Return negative infinity for
checkmate, 0 for stalemate
28
29     # Maximizer's turn (white)
30     if is_white:
31         best_move = Move_with_value(0, 0, NEG_INFINITY)
32         best_first_move = None # Stores the best first move at the highest level
33
34         # Iterate through all legal moves
35         for start, value in validation.generate_legal_moves(chessboard_object,
is_white).items():
36             for end in value:
37                 move = Move(start, end)
38                 # Simulate the move on a new chessboard object
39                 new_chessboard_object = validation.make_move(start, end,
chessboard_object)
40                 # Recursively call MINIMAX for the opponent
41                 new_move = MINIMAX(new_chessboard_object, depth - 1, False, move)
42
43                 # Update the best move if a better one is found
44                 if new_move.value > best_move.value:
```

```

45         best_move = new_move
46         best_first_move = move # Store the first move at the top level
47
48     # If no moves are possible
49     if best_first_move is None:
50         return best_move
51
52     # Return the best first move found at the current level
53     return Move_with_value(best_first_move.start, best_first_move.end,
best_move.value)
54
55     # Minimizer's turn (black)
56 else:
57     best_move = Move_with_value(0, 0, INFINITY)
58     best_first_move = None # Stores the best first move at the highest level
59
60     # Iterate through all legal moves
61     for start, value in validation.generate_legal_moves(chessboard_object,
is_white).items():
62         for end in value:
63             move = Move(start, end)
64             # Simulate the move on a new chessboard object
65             new_chessboard_object = validation.make_move(start, end,
chessboard_object)
66             # Recursively call MINIMAX for the opponent
67             new_move = MINIMAX(new_chessboard_object, depth - 1, True, move)
68
69             # Update the best move if a better one is found
70             if new_move.value < best_move.value:
71                 best_move = new_move
72                 best_first_move = move # Store the first move at the top level
73
74     # If no moves are possible
75     if best_first_move is None:
76         return best_move
77
78     # Return the best first move found at the current level
79     return Move_with_value(best_first_move.start, best_first_move.end,
best_move.value)

```

Hier werden wie bereits erwähnt noch keine Verbesserungen wie Alpha-Beta Pruning, Move Ordering oder Iterative Deepening verwendet.

Ich möchte an dieser Stelle zunächst meine Evaluierungsfunktion auf ein zufriedenstellendes Niveau weiterentwickeln, um anschließend den Suchalgorithmus mit den verschiedenen Verbesserungen zu vergleichen um somit die Effizienz der Verbesserung besser einschätzen zu können.

5 Rochade und En Passant

5.1 Chessboard_state

Um Rochaderechte sowie En Passant Möglichkeiten zu speichern, habe ich das einfache Chessboard durch ein `Chessboard_state` -Objekt ersetzt, die oben beschriebenen Funktionen

bekommen nun also Teilweise nicht mehr nur ein chessboard übergeben, sondern ein solches Chessboard_state -Objekt, welches ein Attribut chessboard besitzt. Genauso geben manche der Funktionen nun ein Chessboard_State -Objekt zurück, um später rekursiv auf diesen die Suchfunktion aufzurufen.

Die Funktionsweise an sich ist unverändert.

```
1 class Chessboard_state:
2     def __init__(self, chessboard, castling, en_passant):
3         self.chessboard = chessboard # 2D array representing the board
4         self.castling = castling # Castling rights
5         self.en_passant = en_passant # En passant target square
```

Dieses wird mit der neuen Funktion fen_to_chessboard_object erstellt.

```
1 # Convert FEN string to a chessboard object
2 def fen_to_chessboard_object(fen):
3     # Split FEN into board position and game information
4     chessboard_fen, information_fen = fen.split(" ",1)
5     chessboard = fen_to_array(chessboard_fen) # Convert the position part to a 2D
        array
6
7     # Extract additional information from FEN
8     turn_right, castling, en_passant, halfmove, fullmove = information_fen.split("
")
9
10    # Create a chessboard state object with the position and castling/en passant
        rights
11    chessboard_object = Chessboard_state(chessboard, castling, en_passant)
12    return chessboard_object
```

Wie zu sehen ist, könnten hier auch noch weitere Informationen gespeichert werden wie

- turn_right
- halfmove
- fullmove

Vielleicht werden diese in Zukunft noch an passender Stelle genutzt.

5.2 Castling

Um legale Rochade-Züge zu berechnen nutzen wir die folgende Funktion:

```
1 # Check if castling is legal and return possible castling moves
2 def legal_castling_moves(possible_castling, chessboard, is_white):
3
4     castling_moves = []
5     if possible_castling == "-": # No castling rights
6         return castling_moves
```

```

7
8     elif is_white: # White castling
9
10         if "Q" in possible_castling: # Queenside castling
11             # Check if squares between king and rook are empty
12             Q_possible = chessboard[7][1] == 0 and chessboard[7][2] == 0 and
chessboard[7][3] == 0
13
14             if Q_possible:
15                 sim_chessboard = copy.deepcopy(chessboard) # Copy Chessboard
16
17                 # Simulate the castling move
18                 sim_chessboard[7][2] = 'K'
19                 sim_chessboard[7][3] = 'R'
20
21                 # Check if castling would leave the king in check
22                 if not is_check(sim_chessboard, is_white):
23                     castling_moves.append((7,2))
24
25         if "K" in possible_castling: # Kingside castling
26             # Check if squares between king and rook are empty
27             K_possible = chessboard[7][5] == 0 and chessboard[7][6] == 0
28
29             if K_possible:
30                 sim_chessboard = copy.deepcopy(chessboard) # Copy Chessboard
31
32                 # Simulate the castling move
33                 sim_chessboard[7][6] = 'K'
34                 sim_chessboard[7][5] = 'R'
35
36                 # Check if castling would leave the king in check
37                 if not is_check(sim_chessboard, is_white):
38                     castling_moves.append((7,6))
39
40     else: # Black castling
41
42         if "q" in possible_castling: # Queenside castling
43             logger.debug("q detected")
44
45             # Check if squares between king and rook are empty
46             Q_possible = chessboard[0][1] == 0 and chessboard[0][2] == 0 and
chessboard[0][3] == 0
47
48             if Q_possible:
49                 sim_chessboard = copy.deepcopy(chessboard) # Copy Chessboard
50
51                 # Simulate the castling move
52                 sim_chessboard[0][2] = 'k'
53                 sim_chessboard[0][3] = 'r'
54
55                 # Check if castling would leave the king in check
56                 if not is_check(sim_chessboard, is_white):
57                     castling_moves.append((0,2))
58
59         if "k" in possible_castling: # Kingside castling
60             # Check if squares between king and rook are empty
61             K_possible = chessboard[0][5] == 0 and chessboard[0][6] == 0

```



```

62
63     if K_possible:
64         sim_chessboard = copy.deepcopy(chessboard) # Copy Chessboard
65
66         # Simulate the castling move
67         sim_chessboard[0][6] = 'k'
68         sim_chessboard[0][5] = 'r'
69
70         # Check if castling would leave the king in check
71         if not is_check(sim_chessboard, is_white):
72             castling_moves.append((0,6))
73
74     return castling_moves

```

Der übergebene Parameter `possible_castling` enthält einen String der Form `"qkQK"`. Das bedeutet in diesem Fall, dass Schwarz auf Dame-Seite ("q") sowie ("k") rochieren darf. Für Weiß gilt in diesem Fall das selbe ("Q" und "K").

5.3 En Passant

Der Ablauf ist bei der Berechnung der legalen En Passant Züge ähnlich zu den legalen Castling Zügen, hier unterscheidet sich nur die Zugberechnung selbst.

```

1  # Check if en passant is legal and return possible en passant moves
2
3  def legal_en_passant_moves(possible_en_passant, chessboard, is_white):
4
5
6
7      en_passant_moves = {}
8
9
10
11     # If no en passant target square is specified in the FEN string
12
13     if possible_en_passant == '-':
14
15         return {}
16
17
18
19     # If an en passant target square is specified
20
21     else:
22
23         # Convert the en passant target square from algebraic notation (e.g.,
24         # 'e6')
25
26         # to a tuple of (row, column) where (0, 0) is the top-left square.
27
28         en_passant_move = list(possible_en_passant)
29
30         column = en_passant_move[0]

```

```

31         converted_column = ord(column) - ord("a") # e.g., 'a' -> 0, 'b' -> 1,
    ..., 'h' -> 7
32
33         converted_row = 7 - (int(en_passant_move[1]) - 1) # e.g., '6' -> row 2
    (0-indexed)
34
35
36
37         converted_en_passant_move = (converted_row, converted_column)
38
39
40
41         # If the en passant target row is the 5th rank (for black's pawn
    capture)
42
43         if converted_row == 5:
44
45             # Check for black pawns that can make the en passant capture
46
47             for field in [-1, +1]: # Check the squares to the left and right of
    the target square
48
49                 # Check if the adjacent square contains a black pawn and is
    within the board boundaries
50
51                 if in_bound(converted_row - 1, converted_column + field) and
    chessboard[converted_row - 1][converted_column + field] == 'p':
52
53
54
55                 # Simulate the en passant move
56
57                 sim_chessboard = copy.deepcopy(chessboard)
58
59                 sim_chessboard[converted_row][converted_column] = 'p' #
    Place the capturing pawn on the target square
60
61                 sim_chessboard[converted_row - 1][converted_column + field]
    = 0
62
63
64
65                 sim_chessboard[converted_row - 1][converted_column] = 0 #
    remove captured white pawn
66
67
68
69                 logger.debug(f"Checking p on {converted_row - 1}
    {converted_column + field} with Field = {field}")
70
71                 logger.debug(sim_chessboard)
72
73                 # Check if the move leaves the black king in check
74
75                 if not is_check(sim_chessboard, is_white):
76

```

```

77             # If the move is legal, add it to the possible en
passant moves
78
79             # The key is the starting position of the capturing
pawn, the value is a list containing the target square
80
81             en_passant_moves.setdefault((converted_row - 1,
converted_column + field), []).append(converted_en_passant_move)
82
83
84
85         # If the en passant target row is the 3rd rank (for white's pawn
capture)
86
87         elif converted_row == 3:
88
89             # Check for white pawns that can make the en passant capture
90
91             for field in [-1, +1]: # Check the squares to the left and right of
the target square
92
93                 # Check if the adjacent square contains a white pawn and is
within the board boundaries
94
95                 if in_bound(converted_row + 1, converted_column + field) and
chessboard[converted_row + 1][converted_column + field] == 'P':
96
97
98
99                 # Simulate the en passant move
100
101                 sim_chessboard = copy.deepcopy(chessboard)
102
103                 sim_chessboard[converted_row][converted_column] = 'P' #
Place the capturing pawn on the target square
104
105                 sim_chessboard[converted_row + 1][converted_column + field]
= 0
106
107
108
109
110                 sim_chessboard[converted_row + 1][converted_column] = 0 #
remove captured black pawn
111
112
113
114             # Check if the move leaves the white king in check
115
116             if not is_check(sim_chessboard, is_white):
117
118                 # If the move is legal, add it to the possible en
passant moves
119
120                 # The key is the starting position of the capturing
pawn, the value is a list containing the target square
121

```

```

122             en_passant_moves.setdefault((converted_row + 1,
        converted_column + field), []).append(converted_en_passant_move)
123
124
125
126     return en_passant_moves

```

- ☐ **TODO** Move_Object erklären und Object mit Value Dings
- ☐ **TODO:** Aktuell move_with_value um value an frontend zu übergeben, nur Move übergeben, value mit evaluateValue im Client berechnen, funktion existiert eh schon (Doch kein Sinn?)

Wichtig hierbei:

Bei dem Erstellen des Suchbaums später, muss nur die Rochade-Rechte an entstehende Kind-States weitergegeben werden, da En Passant nur für einen Halbzug möglich ist, verfallen En Passant Möglichkeiten sofort.

Die Rochade-Rechte jedoch können sich durch Bewegen von Türmen oder des Königs auch in Folgezuständen ändern.

- ☐ **TODO** Rochade Weiterleiten an Kindknoten

6 Evaluation Function

Um die Evaluierungsfunktion nun um die in der theory.md vorgestellten Konzepte zu erweitern, brauchen wir zunächst eine Funktion, welche über das Schachbrett iteriert und für jede Figur, die auf diese Figur zutreffenden Konzepte zu überprüfen.

2.2.2 Center Control

Dynamic Control

- ☒ Implemented

-> Funktioniert, Engine zieht Bauern vermehrt in die Mitte

2.2.3 King Safety

Pawn Shield

- ☒ Implemented

Virtual Mobility

- ☒ Implemented

King Tropism

✓ Implemented

2.2.4 Evaluation of Pieces

Pawn

✓ Implemented

Knight

✓ Implemented

Bishop

✓ Implemented

Rook

✓ Implemented

Queen

✓ Implemented

An dieser Stelle habe ich bereits für den Test meine Engine gegen verschieden starke Bots von Chess.com spielen lassen.

Hier ist mir aufgefallen, dass Remis durch Zugwiederholung noch nicht betrachtet wurde.

Bei vielen Partien sind die Engines in eine Art Schleife gekommen, bei der diese auf Basis ihres deterministischen Algorithmus immer wieder die selben Züge gespielt haben.

7. Improved Searchfunction

Um die Verbesserungen an der Suchfunktion vorzunehmen, habe ich diese zunächst etwas vereinfacht und refactored.

```
1  # Minimax algorithm implementation to find the best move
2
3  def MINIMAX(chessboard_object, depth, is_white, move_leading_here):
4
5      global counter
6
7      counter += 1
8
9      logger.debug(counter)
10
11
12
```

```

13     chessboard = chessboard_object.chessboard
14
15
16
17     # BASE CASES
18
19     game_over, reason = validation.game_over(chessboard_object, is_white)
20
21     if game_over:
22
23         if reason == 0: # Checkmate
24
25             value = INFINITY if is_white else NEG_INFINITY
26
27         else: # Stalemate
28
29             value = 0
30
31
32
33     return Move_with_value(move_leading_here.start, move_leading_here.end,
34                             value, None)
35
36
37     if depth == 0: # Max search depth reached
38
39         return Move_with_value(move_leading_here.start, move_leading_here.end,
40                                 validation.evaluate_position(chessboard),
41                                 chessboard_object.promotion)
42
43
44
45
46
47     if is_white: # Maximizing player (White)
48
49         best_move = Move_with_value(None, None, NEG_INFINITY, None)
50
51
52
53         # Generate and iterate over all legal moves for White
54
55         for start, value_list in
56             validation.generate_legal_moves(chessboard_object, is_white).items():
57
58             for end in value_list:
59
60                 new_chessboard_object = validation.make_move(start, end,
61                     chessboard_object)
62
63                 current_promotion = new_chessboard_object.promotion
64
65                 action_taken = Move(start, end)
66

```

```

67         # Recursive call for Black
68
69         result = MINIMAX(new_chessboard_object, depth - 1, False,
    action_taken)
70
71
72
73         # Update best move if a better one is found
74
75         if result.value > best_move.value:
76
77             best_move = Move_with_value(start, end, result.value,
    current_promotion)
78
79
80
81         return best_move
82
83
84
85     else: # Minimizing player (Black)
86
87         best_move = Move_with_value(None, None, INFINITY, None)
88
89
90
91         # Generate and iterate over all legal moves for Black
92
93         for start, value_list in
    validation.generate_legal_moves(chessboard_object, is_white).items():
94
95             for end in value_list:
96
97                 new_chessboard_object = validation.make_move(start, end,
    chessboard_object)
98
99                 current_promotion = new_chessboard_object.promotion
100
101                 action_taken = Move(start, end)
102
103
104
105         # Recursive call for White
106
107         result = MINIMAX(new_chessboard_object, depth - 1, True,
    action_taken)
108
109
110
111         # Update best move if a worse (lower) value is found
112
113         if result.value < best_move.value:
114
115             value =
    validation.evaluate_position(chessboard_object.chessboard)
116

```

```

117         best_move = Move_with_value(start, end, result.value,
    current_promotion)
118
119
120
121     return best_move

```

Das vereinfacht uns das Erweitern und hält die Funktion übersichtlich.

Besser wäre er gewesen, wenn ich zuerst die Suchfunktion erweitert hätte, da das Fine-Tuning somit deutlich einfacherer gefallen wäre.

○ **TODO** Weiter ausführen

Nun können wir also die weiteren Verbesserungen vornehmen.

An dieser Stelle verweise ich erneut auf die `theory.md`, welche jede dieser Verbesserungen ausführlicher erklärt.

7.1 Alpha-Beta-Pruning

Zunächst betrachten wir wieder einen einfachen Pseudocode:

```

1  function minimax(node, depth, alpha, beta, maximizingPlayer):
2
3      if depth == 0 or node is terminal:
4          return heuristic_value(node)
5
6      if maximizingPlayer:
7          maxEval = -∞
8          for each child in node.children:
9              eval = minimax(child, depth - 1, alpha, beta, false)
10             maxEval = max(maxEval, eval)
11             alpha = max(alpha, eval)
12             if beta <= alpha:
13                 break // Beta cut-off
14             return maxEval
15
16      else: // Minimizing player
17          minEval = +∞
18          for each child in node.children:
19              eval = minimax(child, depth - 1, alpha, beta, true)
20              minEval = min(minEval, eval)
21              beta = min(beta, eval)
22              if beta <= alpha:
23                  break // Alpha cut-off
24          return minEval

```

Der Aufbau bleibt also prinzipiell gleich, wir erweitern jedoch unseren generischen MiniMax-Algorithmus indem wir:

- `alpha, beta` als Parameter hinzufügen

- `alpha` bei Minimizer berechnen
- `beta` bei Maximizer berechnen
- eine Abbruchsbedingung hinzufügen.

```

1  def MINIMAX(chessboard_object, depth, max_depth, is_white, move_leading_here,
2      alpha, beta):
3      global counter
4
5      counter += 1
6
7      logger.debug(counter)
8
9
10
11     chessboard = chessboard_object.chessboard
12
13
14
15     # BASE CASES
16
17     game_over, reason = validation.game_over(chessboard_object, is_white)
18
19
20
21     if game_over:
22
23         logger.debug(f"MATT: {chessboard_object.chessboard}")
24
25         if reason == 0: # Checkmate
26
27             value = -10000 + depth if is_white else 10000 - depth
28
29         else: # Stalemate
30
31             value = 0
32
33         return Move_with_value(move_leading_here.start, move_leading_here.end,
34                                value, None)
35
36
37     if depth == max_depth: # Max search depth reached
38
39         return Move_with_value(move_leading_here.start, move_leading_here.end,
40                                validation.evaluate_position(chessboard),
41                                chessboard_object.promotion)
42
43
44
45
46
47     if is_white: # Maximizing player (White)
48
49         best_move = Move_with_value(None, None, NEG_INFINITY, None)
50

```

```

51         legal_moves = validation.generate_legal_moves(chessboard_object,
is_white)
52
53         ordered_moves = order_moves(chessboard_object, legal_moves, True)
54
55
56
57         for move in ordered_moves:
58
59             new_chessboard_object = validation.make_move(move.start, move.end,
chessboard_object)
60
61             current_promotion = new_chessboard_object.promotion
62
63             action_taken = Move(move.start, move.end)
64
65
66
67             result = MINIMAX(new_chessboard_object, depth + 1, max_depth, not
is_white, action_taken, alpha, beta)
68
69
70
71             if result.value > best_move.value:
72
73                 best_move = Move_with_value(action_taken.start,
action_taken.end, result.value, current_promotion)
74
75
76
77
78                 alpha = max(alpha, best_move.value)
79
80
81
82                 if beta <= alpha:
83
84                     return best_move # Prune
85
86
87
88         return best_move
89
90
91
92     else: # Minimizing player (Black)
93
94
95
96         best_move = Move_with_value(None, None, INFINITY, None)
97
98         legal_moves = validation.generate_legal_moves(chessboard_object,
is_white)
99
100         ordered_moves = order_moves(chessboard_object, legal_moves, False)
101
102

```

```

103
104     for move in ordered_moves:
105
106         new_chessboard_object = validation.make_move(move.start, move.end,
chessboard_object)
107
108         current_promotion = new_chessboard_object.promotion
109
110         action_taken = Move(move.start, move.end)
111
112         result = MINIMAX(new_chessboard_object, depth + 1, max_depth, not
is_white, action_taken, alpha, beta)
113
114
115
116         if result.value < best_move.value:
117
118             best_move = Move_with_value(action_taken.start,
action_taken.end, result.value, current_promotion)
119
120
121
122         beta = min(beta, best_move.value)
123
124
125
126         if beta <= alpha:
127
128             return best_move # Prune
129
130
131
132
133     return best_move

```

Zudem habe ich hier direkt ein einfaches Move Ordering hinzugefügt, somit wird Alpha-Beta-Pruning direkt um einiges effizienter.

7.2 Move Ordering

Bei dem Sortieralgorithmus habe ich mich für QuickSort entschieden.

Quicksort macht Sinn, weil es sehr schnell, leichtgewichtig und gut für kleine bis mittlere Listen ist, also perfekt für die Sortierung von Schachzügen.

Genauer nehme ich eine einfachere Evaluierungsfunktion um die Züge bereits vor zu evaluieren und zu sortieren. Diese einfache Evaluierungsfunktion haben wir schon bereits genutzt, als erste, einfache Evaluierungsfunktion:

```

1  # Returns the value of a given move
2
3  def get_move_value(start, end, chessboard_object):
4

```

```

5     sim = validation.make_move(start, end, chessboard_object) # Simulates the
      move
6
7     return validation.calc_piece_value_with_psqd(sim.chessboard) # Calculates
      the heuristic value

```

Die genaue Implementierung ist in der `searchfunction.py` zu finden.

7.3 Iterative Deepening

☐ TODO

Bekannte Probleme / Bug fixes

- ☐ Bei erzwungenem Matt gibt Backend für Engine keinen validen Zug zurück
- ☐ Wenn 3VAL verliert wird der Roboter nicht richtig aktualisiert

Bug Fixes

Engine setzt dich nicht Schachmatt im Endgame.

Problem identifiziert: Engine hat nicht zwischen schnellen Matt und Matt in tieferer Tiefe unterschieden -> Fehler bei Bewertung der Züge.

Lösung: Gebe bzw. ziehe Punkte ab für früheres Matt.

Problem Identifizierung + Lösung: 5 Stunden

Geplante Verbesserungen

Kritische Korrekturen

- **Zustandsaktualisierung in `make_move`** : Stelle sicher, dass `make_move` nicht nur das Brett aktualisiert, sondern auch die **Rochaderechte** und das **En-Passant-Zielfeld** korrekt anpasst, wenn Züge dies erfordern. Dies ist entscheidend für die Korrektheit.

Effizienz

- **`make_move` in der Suche:** Erwäge statt `copy.deepcopy()` einen "Make/Unmake"-Ansatz (Zug ausführen -> Rekursion -> Zug zurücknehmen), um die Performance der **MINIMAX**-Suche zu verbessern.
- **`generate_legal_moves`** : Optimierte die Prüfung auf legale Züge. Statt jeden Zug zu simulieren (`make_move`) und dann `is_check` aufzurufen, prüfe gezielter, ob der König nach dem Zug im Schach stehen würde (z.B. durch Analyse von Fesselungen und Angreifen).

Code-Struktur & Design (OOP & DRY)

- **Code-Duplizierung (DRY - Don't Repeat Yourself):**

- Vereinheitliche die Zuggenerierungslogik für weiße und schwarze Figuren (z.B. in `generate_moves` , `dynamic_control`).
- Abstrahiere die Logik für gleitende Figuren (Läufer, Turm, Dame) in eine gemeinsame Hilfsfunktion.
- Vermeide wiederholte Berechnungen (z.B. `king_tropism` Aufrufe in `evaluate_position`).
- **Objektorientierung (OOP):**
 - Baue die `Chessboard_state` -Klasse weiter aus. Sie könnte Methoden wie `make_move()` , `unmake_move()` , `get_legal_moves()` , `is_check()` , `evaluate()` enthalten, um Zustand und Verhalten stärker zu kapseln.

Robustheit & API

- **FEN-Parsing:** Füge Fehlerbehandlung hinzu, um ungültige FEN-Strings abzufangen.
- **Fehlerbehandlung:** Nutze Python Exceptions statt spezieller Rückgabewerte (wie `-1`) für Fehlerfälle.
- **API-Antworten:** Gib von den API-Endpunkten standardmäßig JSON zurück, statt reiner Strings (besonders bei `/evaluate-position`).
- **Globale Zustände:** Vermeide das Verändern globaler Variablen (z.B. `searchfunction.counter`) direkt in API-Routen, um Probleme bei gleichzeitigen Anfragen zu verhindern.

Lesbarkeit & Wartbarkeit

- **Lange Funktionen:** Teile sehr lange Funktionen (z.B. für Rochade, En Passant, komplexe Bewertungsteile) in kleinere, fokussierte Hilfsfunktionen auf.