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.

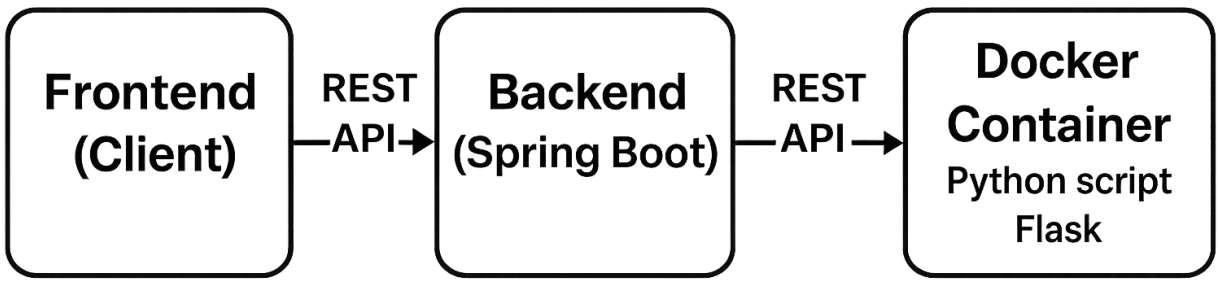
Beim Styling habe ich mich für den BEM-Syntax entschieden, um ein besseres Verständnis für CSS-Prinzipien zu erlangen und gleichzeitig eine professionelle Benennung meiner CSS Elemente zu garantieren.

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 Evaluierungfunktion 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

```
volumes:
  - ./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 Begin 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.

```
# Convert FEN into 2D Array
def fen_to_array(fen):

  # 2D Array 8x8
  chessboard = [[0 for _ in range(8)] for _ in range(8)]

  current_field = 0
  current_row = 0
  current_column = 0

  for char in fen:

    if char.isalpha():
      current_row = current_field // 8
      current_column = current_field % 8
      chessboard[current_row][current_column] = char
      current_field += 1

    elif char.isdigit():
      current_field += int(char)

    elif char == " ":
      break

  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:

```
class piece(IntEnum):
  QUEEN = 900
  ROOK= 400
  BISHOP = 300
  KNIGHT = 300
  PAWN = 100
  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 Stockfisch, 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 Spielbegin festgelegt:

```
class Psqt:

  PAWN_PSQT = [
    0,   0,   0,   0,   0,   0,   0,   0,
    98, 134,  61,  95,  68, 126,  34, -11,
    -6,   7,  26,  31,  65,  56,  25, -20,
    -14,  13,   6,  21,  23,  12,  17, -23,
    -27,  -2,  -5,  12,  17,   6,  10, -25,
    -26,  -4,  -4,  -2,   4,   4,  -4, -24,
    -35, -17, -13, -10, -11, -13, -17, -37,
    0,   0,   0,   0,   0,   0,   0,   0
  ]

  KNIGHT_PSQT = [
    -167, -89, -34, -49,  61, -97, -15, -107,
    -73,  -41,  72,  36,  23,  62,   7,  -17,
    -47,  60,  37,  65,  84, 129,  73,   44,
    -9,  17,  19,  53,  37,  69,  18,   22,
    -13,   4,  16,  13,  28,  19,  21,  -8,
    -23,  -9,  12,  10,  19,  17,  25, -16,
    -29, -53, -12,  -3,  -1,  18, -14,  -19,
    -105, -21, -58, -33, -17, -28, -19,  -23
  ]

  BISHOP_PSQT = [
    -29,   4, -82, -37, -25, -42,   7,  -8,
    -26,  16, -18, -13,  30,  59,  18, -47,
    -16,  37,  43,  40,  35,  50,  37,  -2,
    -4,   5,  19,  50,  37,  37,   7,  -2,
    2,  -8,  16,  29,  27,  41,  12,   9,
    -6,   3,   9,  23,  24,  35,   6,  10,
    -6,   0,   4,  -7,  -1,  -5,  24, -18,
    -17, -17, -14,  -8,  -6, -10, -15, -21
  ]

  ROOK_PSQT = [
    32,  42,  32,  51,  63,   9,  31,  43,
    27,  32,  58,  62,  80,  67,  26,  44,
    -5,  19,  26,  36,  17,  45,  61,  16,
    -24,  -4,   4,   1,  -7,  -6,  -9, -16,
    -36, -27, -20, -12,  -1,  -7,   3, -37,
    -39, -18,  -9,   3,   6,  -6,  -3, -45,
    -44, -16, -20,  -9,  -1,  11,  -6, -71,
```

```
    -19, -13,   1,  17,  16,   7, -37, -26
  ]

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

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

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:

```python
# Get the position value from the piece-square table for a given piece and position

def get_psqt_value(field, piece, is_white):



    psqt = get_psqt_table(piece, is_white)  # Get the appropriate piece-square table

    kingOrQueen = piece.upper() == "K" or piece.upper() == "Q"

    # For black pieces or kings/queens, use the mirrored square value

    if not is_white or kingOrQueen:

        return psqt[63 - field]  # Return mirrored value

    else:

         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.

```
...

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

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

...
```

Diese sollen später noch um solche für Midgame und Endgame erweitert werdeb.

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.

```
def MINIMAX(position, depth, isMaximizingPlayer):
  if depth == 0 or game_over(position):
    return evaluate(position)  # Bewertung der Stellung

  if isMaximizingPlayer:
    bestValue = -∞
    for each child in generate_moves(position):
      value = MINIMAX(child, depth - 1, false)
      bestValue = max(bestValue, value)
    return bestValue

  else:
    bestValue = ∞
    for each child in generate_moves(position):
      value = MINIMAX(child, depth - 1, true)
      bestValue = min(bestValue, value)
    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.
Diese werden der Einfachheit halber hier jedoch nicht weiter thematisiert.

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.

```
def generate_moves(chessboard, is_white):

  moves = {} # Dic for all moves

  for i in range(8):
    for j in range(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:

```
def pawn_moves(field_row, field_column, is_white, chessboard):

  possible_moves = []

  if is_white:
    forward_row = field_row - 1
    start_row = 6
    direction = -1  # Weiß zieht nach oben
  else:
    forward_row = field_row + 1
    start_row = 1
    direction = 1  # Schwarz zieht nach unten

  # Ein Feld vorwärts, wenn frei
  if chessboard[forward_row][field_column] == 0:
    possible_moves.append((forward_row, field_column))

    # Zwei Felder vorwärts, wenn Startposition und beide Felder frei
    if field_row == start_row and chessboard[forward_row + direction]
[field_column] == 0:
      possible_moves.append((forward_row + direction, field_column))

  # Schlagzüge nach links und rechts prüfen
  for side in [-1, 1]:  # -1 = links, +1 = rechts
    new_column = field_column + side
    if 0 <= new_column <= 7:  # Prüfen, ob innerhalb der Spielfeldgrenzen
      target_piece = chessboard[forward_row][new_column]
      if target_piece != 0 and is_enemy(is_white, target_piece):  # Feindliche
Figur dort?
        possible_moves.append((forward_row, new_column))

  return possible_moves

```

En 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:

```
def queen_moves(field_row, field_column, is_white, chessboard):

```

```
    move_pattern = constants.Piece_moves.Queen
    possible_moves = []

    for direction in move_pattern:
      row, column = field_row, field_column  # Startposition

      while in_bound(row + direction[0], column + direction[1]):

        row += direction[0] # directioion[0] stores vertical movement
        column += direction[1] # directioion[0] stores horizontal movement

        if chessboard[row][column] == 0:
          possible_moves.append((row, column))

        elif chessboard[row][column] != 0 and is_enemy(is_white, chessboard[row][column]):
          possible_moves.append((row, column))
          break  # Gegner blockiert weitere Bewegung

        else: # Eigene Figur blockiert Bewegung
          break

    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 En Passant und Rochade

4.2 Matt erkennen

Die Matt-Erkennung funktioniert nach dem folgenden Schema:

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

```
def is_check(chessboard, is_white):

  all_opponent_moves = generate_moves(chessboard, not is_white) # Returns all possible moves of opponent
  king_field = get_king_field(chessboard, is_white)

  for key, value in all_opponent_moves.items(): # Iterate trough all opponents moves
     if (len(value) != 0):
       for move in value:

         if move == king_field: # King is attacked -> check
           return True
  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

```
def game_over(chessboard, is_white):

  legal_moves = len(generate_legal_moves(chessboard, is_white))

  if legal_moves != 0:
    return False, None
  elif is_check:
    return True, 0
  else:
    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:

```
# Minimax algorithm implementation to find the best move
def MINIMAX(chessboard_object, depth, is_white, move):
  global counter
  counter += 1
  logger.debug(counter)

  chessboard = chessboard_object.chessboard

  logger.debug(f"MINMAX SEARCH: {counter}")
  logger.debug(chessboard)

  # Base case: if the maximum depth is reached
  if depth == 0:
    # Check if the game is over (checkmate or stalemate)
    game_over, reason = validation.game_over(chessboard_object, is_white)

    # If the game is not over, evaluate the current position
    if not game_over: # not Checkmate or Stalemate
      return Move_with_value(move.start, move.end,
validation.evaluate_position(chessboard)) # Evaluate the position

    # If the game is over and it's the maximizer's turn (white)
    if is_white: # Checkmate or Stalemate and Maximizer
      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

    # If the game is over and it's the minimizer's turn (black)
    else: # Checkmate or Stalemate and Minimizer
```

```
      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

  # Maximizer's turn (white)
  if is_white:
    best_move = Move_with_value(0, 0, NEG_INFINITY)
    best_first_move = None # Stores the best first move at the highest level

    # Iterate through all legal moves
    for start, value in validation.generate_legal_moves(chessboard_object, is_white).items():
      for end in value:
        move = Move(start, end)
        # Simulate the move on a new chessboard object
        new_chessboard_object = validation.make_move(start, end, chessboard_object)
        # Recursively call MINIMAX for the opponent
        new_move = MINIMAX(new_chessboard_object, depth - 1, False, move)

        # Update the best move if a better one is found
        if new_move.value > best_move.value:
          best_move = new_move
          best_first_move = move # Store the first move at the top level

    # If no moves are possible
    if best_first_move is None:
      return best_move

    # Return the best first move found at the current level
    return Move_with_value(best_first_move.start, best_first_move.end, best_move.value)

  # Minimizer's turn (black)
  else:
    best_move = Move_with_value(0, 0, INFINITY)
    best_first_move = None # Stores the best first move at the highest level

    # Iterate through all legal moves
    for start, value in validation.generate_legal_moves(chessboard_object, is_white).items():
      for end in value:
        move = Move(start, end)
        # Simulate the move on a new chessboard object
        new_chessboard_object = validation.make_move(start, end, chessboard_object)
        # Recursively call MINIMAX for the opponent
        new_move = MINIMAX(new_chessboard_object, depth - 1, True, move)

        # Update the best move if a better one is found
        if new_move.value < best_move.value:
          best_move = new_move
          best_first_move = move # Store the first move at the top level

    # If no moves are possible
    if best_first_move is None:
      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 bleibt 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:

```
# Check if castling is legal and return possible castling moves
def legal_castling_moves(possible_castling, chessboard, is_white):

  castling_moves = []
  if possible_castling == "-":  # No castling rights
    return castling_moves

  elif is_white:  # White castling

    if "Q" in possible_castling:  # Queenside castling
      # Check if squares between king and rook are empty
      Q_possible = chessboard[7][1] == 0 and chessboard[7][2] == 0 and
chessboard[7][3] == 0

      if Q_possible:
        sim_chessboard = copy.deepcopy(chessboard)  # Copy Chessboard

        # Simulate the castling move
        sim_chessboard[7][2] = 'K'
        sim_chessboard[7][3] = 'R'

        # Check if castling would leave the king in check
        if not is_check(sim_chessboard, is_white):
          castling_moves.append((7,2))

    if "K" in possible_castling:  # Kingside castling
      # Check if squares between king and rook are empty
      K_possible = chessboard[7][5] == 0 and chessboard[7][6] == 0

      if K_possible:
        sim_chessboard = copy.deepcopy(chessboard)  # Copy Chessboard

        # Simulate the castling move
        sim_chessboard[7][6] = 'K'
        sim_chessboard[7][5] = 'R'

        # Check if castling would leave the king in check
        if not is_check(sim_chessboard, is_white):
          castling_moves.append((7,6))

  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 unterschiedet 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
```

```
    # If no en passant target square is specified in the FEN string
    if possible_en_passant == '-':
        return {}

    # If an en passant target square is specified
    else:
        # Convert the en passant target square from algebraic notation (e.g., 'e6')
        # to a tuple of (row, column) where (0, 0) is the top-left square.
        en_passant_move = list(possible_en_passant)
        column = en_passant_move[0]
        converted_column = ord(column) - ord("a")  # e.g., 'a' -> 0, 'b' -> 1, ..., 'h' -> 7
        converted_row = 7 - (int(en_passant_move[1]) - 1) # e.g., '6' -> row 2 (0-indexed)

        converted_en_passant_move = (converted_row, converted_column)

        # If the en passant target row is the 5th rank (for black's pawn capture)
        if converted_row == 5:
            # Check for black pawns that can make the en passant capture
            for field in [-1, +1]: # Check the squares to the left and right of the target square
                # Check if the adjacent square contains a black pawn and is within the board boundaries
                if in_bound(converted_row - 1, converted_column + field) and chessboard[converted_row - 1][converted_column + field] == 'p':

                    # Simulate the en passant move
                    sim_chessboard = copy.deepcopy(chessboard)
```

```
                    sim_chessboard[converted_row][converted_column] = 'p' # Place the capturing pawn on the target square

                    sim_chessboard[converted_row - 1][converted_column + field] = 0



                    sim_chessboard[converted_row - 1][converted_column] = 0 # remove captured white pawn



                    logger.debug(f"Checking p on{converted_row - 1} {converted_column + field} with Field = {field}")

                    logger.debug(sim_chessboard)

                    # Check if the move leaves the black king in check

                    if not is_check(sim_chessboard, is_white):

                        # If the move is legal, add it to the possible en passant moves

                        # The key is the starting position of the capturing pawn, the value is a list containing the target square

                        en_passant_moves.setdefault((converted_row - 1, converted_column + field), []).append(converted_en_passant_move)



        # If the en passant target row is the 3rd rank (for white's pawn capture)

        elif converted_row == 3:

            # Check for white pawns that can make the en passant capture

            for field in [-1, +1]: # Check the squares to the left and right of the target square

                # Check if the adjacent square contains a white pawn and is within the board boundaries

                if in_bound(converted_row + 1, converted_column + field) and chessboard[converted_row + 1][converted_column + field] == 'P':



                    # Simulate the en passant move

                    sim_chessboard = copy.deepcopy(chessboard)

                    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

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.

```
1   # Evaluates current chessboard position
2
3   def evaluate_position(chessboard):  # pragma: no cover
4
5
```

```
    value = 0

    value += calc_piece_value_with_psqt(chessboard)  # Calculate value based on pieces and their positions TODO integrate in for loop



    black_king_position = get_king_field(chessboard, False)

    white_king_position = get_king_field(chessboard, True)



    pawn_counter = 0 # Keep track of pawns on field for estimation of game progress

    white_rooks = 0

    black_rooks = 0


    for i in range(8):

        for j in range (8):


            piece = chessboard[i][j]


            if piece != 0: # Only evaluate relevant fields (no empty fields)



                # Keep track of pawns on field for estimation of game progress

                if piece.lower() == 'p':

                    pawn_counter += 1


                    # King Tropism


                    distance = distance_to_king(i, j, white_king_position) if piece.islower() else distance_to_king(i, j, black_king_position)

                    value += king_tropism(distance, piece)


```

```
60
61
62                  # Keep track of rooks on field
63
64
65
66                  if piece.lower() == 'r':
67
68
69
70                      if piece.isupper():
71
72                          white_rooks += 1
73
74                      else:
75
76                          black_rooks +=1
77
78
79
80                      # King Tropism
81
82
83
84                      distance = distance_to_king(i, j, white_king_position) if
    piece.islower() else distance_to_king(i, j, black_king_position)
85
86                      value += king_tropism(distance, piece)
87
88
89
90
91
92
93                  # Center fields - Dynamic Control
94
95
96
97                  if 2 < i < 5 and 2 < j < 5:
98
99                      value += constants.Evaluation_factors.DYNAMIC_CONTROL *
    dynamic_control(piece, chessboard, i, j)
100
101
102
103
104                 # King Safety
105
106
107
108                 if piece.lower() == 'k': # King detected
109
110                     value += king_safety(chessboard, piece.isupper(), i, j)
111
112
113
114
```

```
                # EVALUATION OF PIECES



                # Outpost - Knight defended by pawn


                if piece.lower() == 'n': # Knight detected

                    value += constants.Evaluation_factors.KNIGHT_OUTPOST * knight_outpost(chessboard, piece.isupper(), i, j)


                    distance = distance_to_king(i, j, white_king_position) if piece.islower() else distance_to_king(i, j, black_king_position)

                    value += king_tropism(distance, piece)


                # Bad Bishop - Bishop blocked by own pawns


                if piece.lower() == 'b': # Bishop detected

                    value += constants.Evaluation_factors.BAD_BISHOP * bad_bishop(chessboard, piece.isupper(), i, j)


                    distance = distance_to_king(i, j, white_king_position) if piece.islower() else distance_to_king(i, j, black_king_position)

                    value += king_tropism(distance, piece)


                # Queen early development penalty


                if piece.lower() == 'q': # Queen detected

                    value += early_queen_development_penalty(chessboard, piece.isupper(), pawn_counter)


                    distance = distance_to_king(i, j, white_king_position) if piece.islower() else distance_to_king(i, j, black_king_position)

                    value += king_tropism(distance, piece)

```

```
166                     #logger.debug(f"{piece} Tropism value:
    {king_tropism(distance, piece)}")
167
168
169
170
171
172         # Increase rook value depending on pawns on field
173
174
175
176         value += white_rooks * (14 - pawn_counter) ;
177
178         value -= black_rooks * (14 - pawn_counter)
179
180
181
182
183         return value
```

Aktuell wird `calc_piece_value_with_psqt` noch in einer separaten Schachbrett Iteration berechnen, später soll dieser Wert natürlich in der selben Iteration berechnet werden, in der auch die Wertung der verschiedenen Konzepte vorgenommen wird.

`black_king_position, white_king_position, white_rooks, black_rooks, pawn_counter` werden gespeichert und später zu Bewertung bestimmter Konzepte wiederverwendet.

- `black_king_position` -> Position des schwarzen Königs
- `white_king_position` -> Position des weißen Königs
- `white_rooks` -> Anzahl an weißen Türmen
- `black_rooks` -> Anzahl an schwarzen Türmen
- `pawn_counter` -> Anzahl an Pawns (Spiegelt ungefähren Spielfortschritt wider)

2.2.2 Center Control

Dynamic Control

Hier überprüfen wir ob das betrachtete Feld eines der zentralen vier Feldern ist.
Anschließend berechnen wir alle möglichen Züge welche eine Figur auf diesem Feldern machen kann und geben die Anzahl dieser zurück.
`constants.Evaluation_factors.DYNAMIC_CONTROL` ist der Faktor, welcher die relative Bewertung dieser möglichen Züge beschreibt.

```
1   def dynamic_control(piece, chessboard, row, column):
2
3
4
5       is_white_figure = piece.isupper()
6
7       possible_moves = [] # All possible moves for figure on field (row, column)
8
```

```
    if is_white_figure:

        match piece:
            case 'P': possible_moves.extend(pawn_moves(row, column, True, chessboard))
            case 'B': possible_moves.extend(bishop_moves(row, column, True, chessboard))
            case 'N': possible_moves.extend(knight_moves(row, column, True, chessboard))
            case 'R': possible_moves.extend(rook_moves(row, column, True, chessboard))
            case 'Q': possible_moves.extend(queen_moves(row, column, True, chessboard))
            case 'K': possible_moves.extend(king_moves(row, column, True, chessboard))


    else:

        match piece:
            case 'p': possible_moves.extend(pawn_moves(row, column, False, chessboard))
            case 'b': possible_moves.extend(bishop_moves(row, column, False, chessboard))
            case 'n': possible_moves.extend(knight_moves(row, column, False, chessboard))
            case 'r': possible_moves.extend(rook_moves(row, column, False, chessboard))
            case 'q': possible_moves.extend(queen_moves(row, column, False, chessboard))
            case 'k': possible_moves.extend(king_moves(row, column, False, chessboard))

    return len(possible_moves) if is_white_figure else -len(possible_moves)
```

2.2.3 King Safety

Der allgemeine King Safety Wert wird durch folgende Funktion berechnet:

```
# KING SAFETY

def king_safety(chessboard, is_white, row, column):



    king_safety_value = 0

    king_safety_value += constants.Evaluation_factors.KING_SAFETY_PAWN_SHIELD *
pawn_shield(chessboard, is_white, row, column)

    king_safety_value +=
constants.Evaluation_factors.KING_SAFETY_VIRTUAL_MOBILITY *
virtual_mobility(chessboard, is_white, row, column)



    return king_safety_value
```

Die folgenden Funktionen sind dabei selbsterklärend.

Pawn Shield

```
def pawn_shield(chessboard, is_white, row, column):



    pawn_shield_value = 0



    if is_white:

        king_front_row = row -1

        pawn = 'P'

    else:

        king_front_row = row + 1

        pawn = 'p'



    for pawn_position in [1,0,-1]: # Check relevant fields vor pawns



        if in_bound(king_front_row, column + pawn_position):

```

```
30
31
32              if chessboard[king_front_row][column + pawn_position] == pawn:
33
34                  pawn_shield_value += 1
35
36
37
38
39      return pawn_shield_value if is_white else -pawn_shield_value
```

Virtual Mobility

```
1   def virtual_mobility(chessboard, is_white, row, column):
2
3
4
5       possible_queen_moves = []
6
7       possible_queen_moves = queen_moves(row, column, is_white, chessboard)
8
9
10
11      return -len(possible_queen_moves) if is_white else len(possible_queen_moves)
```

King Tropism

```
1   def king_tropism(distance, piece):
2
3
4
5       piece_type = piece.lower()
6
7       is_white = piece.isupper()
8
9
10
11      factors = {
12
13          'q': 100,  # Queen: Sehr hoher Einfluss
14
15          'r': 80,   # Rook: Hoher Einfluss
16
17          'b': 60,   # Bishop: Mittlerer Einfluss
18
19          'n': 50,   # Knight: Mittlerer Einfluss, kann in kurzer Distanz sehr
    wirksam sein
20
21          'p': 20    # Pawn: Geringer Einfluss, kann in direkter Nähe gefährlich
    werden
22
23      }
24
25
```

```
        value = round(factors.get(piece_type) / distance)

        return value if is_white else -value
```

King Tropism ist das einzige Konzept, welches hier nicht direkt in den Wert der King Safety mit einfließt.
Das liegt daran, dass ich mich hier doch für eine etwas andere Implementierung entschiedene habe. Es wird nicht, wie eigentlich geplant, die Distanz zum eigenen König **und** des gegnerischen Königs betrachtet, sondern nur die Distanz zum gegnerischen König, um etwas Druck aufzubauen.
Diese Entscheidung folgte aus der Beobachtung des Spielstils der Engin, diese spielt nach meiner Beobachtung bereits ausreichend defensiv, durch Konzepte wie Pawn Shield, Virtual Mobility und die Strafe für das frühe Entwickeln der Dame.
Stattdessen benutze ich King Tropism hier also um Druck auf den generischen König aufzubauen,

2.2.4 Evaluation of Pieces

Knight

```
# Returns 1 or -1 if knight is defended by pawn depending on color, else 0

def knight_outpost(chessboard, is_white, row, column):



    if is_white:

        behind_row = row + 1

        pawn = 'P'

    else:

        behind_row = row - 1

        pawn = 'p'



    for pawn_position in [1, -1]:


        if in_bound(behind_row, column + pawn_position) and
chessboard[behind_row][column + pawn_position] == pawn:

            return 1 if is_white else -1

    return 0
```

Bishop

```
# Returns number of pawns blocking bishop

def bad_bishop(chessboard, is_white, row, column):



    blocking_pawns = 0



    if is_white:

        front_row = row - 1

        pawn = 'P'

    else:

        front_row = row + 1

        pawn = 'p'




    for pawn_position in [1, -1]:



        if in_bound(front_row, column + pawn_position) and chessboard[front_row][column + pawn_position] == pawn:

            blocking_pawns += 1




    return -blocking_pawns if is_white else blocking_pawns
```

Rook

```
# Increase rook value depending on pawns on field

    value += white_rooks * (14 - pawn_counter)

    value -= black_rooks * (14 - pawn_counter)
```

Queen

```
# Returns penalty for early queen development depending on number of pawns

def early_queen_development_penalty(chessboard, is_white, pawn_counter):
```

```
    penalty = constants.Evaluation_factors.QUEEN_EARLY_DEVELOPMENT_PENALTY


    if is_white:
        queen_row = 7
        queen = 'Q'
    else:
        queen_row = 0
        queen = 'q'


    if pawn_counter >= 14 and chessboard[queen_row][3] != queen: # Opening game - full penalty
        logger.debug("You")
        return -penalty if is_white else penalty
    elif pawn_counter >= 10 and chessboard[queen_row][3] != queen: # Half pentalty
        return -(penalty / 2) if is_white else (penalty / 2)


        # Else no penalty


    else:
        return 0
```

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.
Darum muss sich später noch gekümmert werden.
Da die Engine jedoch sonst schon relativ solides Schach spielt und im Bezug auf das Ziel dieses Projektes, habe ich mich dazu entschieden diese Tatsache vorerst zu ignorieren und mich weiter auf die Suchfunktion zu konzentrieren.

7. Improved Searchfunction

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

```
# Minimax algorithm implementation to find the best move

def MINIMAX(chessboard_object, depth, is_white, move_leading_here):

    global counter

    counter += 1

    logger.debug(counter)



    chessboard = chessboard_object.chessboard



    # BASE CASES

    game_over, reason = validation.game_over(chessboard_object, is_white)

    if game_over:

        if reason == 0:  # Checkmate

            value = INFINITY if is_white else NEG_INFINITY

        else:  # Stalemate

            value = 0



        return Move_with_value(move_leading_here.start, move_leading_here.end,
value, None)



    if depth == 0:  # Max search depth reached

        return Move_with_value(move_leading_here.start, move_leading_here.end,

                               validation.evaluate_position(chessboard),

                               chessboard_object.promotion)



    if is_white:  # Maximizing player (White)

```

```
        best_move = Move_with_value(None, None, NEG_INFINITY, None)


        # Generate and iterate over all legal moves for White
        for start, value_list in validation.generate_legal_moves(chessboard_object, is_white).items():
            for end in value_list:
                new_chessboard_object = validation.make_move(start, end, chessboard_object)
                current_promotion = new_chessboard_object.promotion
                action_taken = Move(start, end)


                # Recursive call for Black
                result = MINIMAX(new_chessboard_object, depth - 1, False, action_taken)


                # Update best move if a better one is found
                if result.value > best_move.value:
                    best_move = Move_with_value(start, end, result.value, current_promotion)


        return best_move


    else:  # Minimizing player (Black)
        best_move = Move_with_value(None, None, INFINITY, None)


        # Generate and iterate over all legal moves for Black
        for start, value_list in validation.generate_legal_moves(chessboard_object, is_white).items():
            for end in value_list:
                new_chessboard_object = validation.make_move(start, end, chessboard_object)
                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.

An dieser Stelle ist mir später aufgefallen, dass es besser gewesen wäre, wenn ich zuerst die Suchfunktion erweitert hätte, da das Fine-Tuning somit deutlich einfacherer gefallen wäre. Habe mich jedoch ursprünglich für diesen Weg entschieden.

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
```

```
        return maxEval

    else:  // Minimizing player
        minEval = +∞
        for each child in node.children:
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  // Alpha cut-off
        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 berrechen
- `beta` bei Maximizer berrechnen
- eine Abbruchsbedingung hinzufügen.

```python
def MINIMAX(chessboard_object, depth, max_depth, is_white, move_leading_here, alpha, beta):

    global counter

    counter += 1

    logger.debug(counter)



    chessboard = chessboard_object.chessboard



    # BASE CASES

    game_over, reason = validation.game_over(chessboard_object, is_white)



    if game_over:

        logger.debug(f"MATT: {chessboard_object.chessboard}")

        if reason == 0:  # Checkmate

            value = -10000 + depth if is_white else 10000 - depth

        else:  # Stalemate

            value = 0

```

```
        return Move_with_value(move_leading_here.start, move_leading_here.end, value, None)


    if depth == max_depth:  # Max search depth reached
        return Move_with_value(move_leading_here.start, move_leading_here.end,
                               validation.evaluate_position(chessboard),
                               chessboard_object.promotion)


    if is_white:  # Maximizing player (White)
        best_move = Move_with_value(None, None, NEG_INFINITY, None)
        legal_moves = validation.generate_legal_moves(chessboard_object, is_white)
        ordered_moves = order_moves(chessboard_object, legal_moves, True)


        for move in ordered_moves:
            new_chessboard_object = validation.make_move(move.start, move.end, chessboard_object)
            current_promotion = new_chessboard_object.promotion
            action_taken = Move(move.start, move.end)


            result = MINIMAX(new_chessboard_object, depth + 1, max_depth, not is_white, action_taken, alpha, beta)


            if result.value > best_move.value:
                best_move = Move_with_value(action_taken.start, action_taken.end, result.value, current_promotion)


            alpha = max(alpha, best_move.value)


            if beta <= alpha:
                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:

```
# Returns the value of a given move

def get_move_value(start, end, chessboard_object):

    sim = validation.make_move(start, end, chessboard_object)  # Simulates the move

    return validation.calc_piece_value_with_psqt(sim.chessboard)  # Calculates the heuristic value
```

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

7.3 Iterative Deepening

◯ TODO

---

8 Weitere Verbesserungen

Dieser Teil befasst sich nun mit weiteren Verbesserungen die ich an meiner Engine vorgenommen habe. Die Grundlegenden Konzepte des Suchalgorithmus und der Evaluierungsfunktion wurden bereits umgesetzt. Hier geht es nun darum, diese noch effizienter zu machen und unnötige Berechnungen zu vermeiden.

8.1 `is_check`

Bei der Berechnung von `is_check` wird aktuell noch jeder Zug des Gegners berechnet und anschließend geschaut, ob einer dieser Züge den eigenen König angreift,

- wenn ja → kein Legaler Zug
- wenn nein → legaler Zug

Dabei könnte man hier viele unnötige Berechnungen verhindern, indem nur die Anwesenheit von Figuren auf bestimmten Felder geprüft wird. So prüfen wir zum Beispiel die offenen Diagonalen des Königs auf Bishops oder eine Queen, die offenen Graden werden auf Rooks oder Queens überprüft. Die den König direkt umliegenden Felder müssen zudem zusätzlich auf Pawns oder einen gegnerischen König überprüft werden.
Auch die Felder auf denen potenziell gefährliche Knight positioniert werden können, können wir gezielt überprüfen.

Figur	Was wird geprüft
Dame	Diagonalen + Linien (wie Läufer + Turm)
Läufer	Diagonalen
Turm	Reihen und Spalten
Springer	8 mögliche Felder mit Sprungmuster
Bauer	Schlagfelder
König	8 Nachbarfelder

```
# Check if the king of the given color is in check

def is_check(chessboard, is_white):



    king_field = get_king_field(chessboard, is_white)  # Find the king's position (Tupel)



    #
    # Diagonal lines (Check for queen or bishop)
    #

    move_pattern = constants.Piece_moves.BISHOP  # Get bishop moves (Check diagionals)


    for direction in move_pattern:

        row, column = king_field[0], king_field[1]  # Starting position


        # Continue moving in the direction until blocked

        while in_bound(row + direction[0], column + direction[1]):


            row += direction[0]  # Vertical movement

            column += direction[1]  # Horizontal movement


```

```
42
43              if chessboard[row][column] != 0:  # Found piece
44
45                  if is_enemy(is_white, chessboard[row][column]) and
    (chessboard[row][column].upper() == "B" or chessboard[row][column].upper() ==
    "Q"): # Enemy bishop or queen found -> Checkmate
46
47                      return True
48
49                  else: # No enemy bishop or queen found
50
51                      break
52
53
54
55
56      #
57      # Vertical and horizontal lines (Check for queen or rook)
58      #
59
60
61
62
63      move_pattern = constants.Piece_moves.ROOK  # Get rook moves (Check
    diagionals)
64
65
66
67      for direction in move_pattern:
68
69          row, column = king_field[0], king_field[1]  # Starting position
70
71
72
73          # Continue moving in the direction until blocked
74
75          while in_bound(row + direction[0], column + direction[1]):
76
77
78
79              row += direction[0]  # Vertical movement
80
81              column += direction[1]  # Horizontal movement
82
83
84
85              if chessboard[row][column] != 0:  # Found piece
86
87                  if is_enemy(is_white, chessboard[row][column]) and
    (chessboard[row][column].upper() == "R" or chessboard[row][column].upper() ==
    "Q"): # Enemy rook or queen found -> Checkmate
88
89                      return True
90
91                  else: # No enemy rook or queen found
92
93                      break
```

```
94
95
96
97
98
99
100         #
101         # Knight moves
102         #
103
104
105
106
107         move_pattern = constants.Piece_moves.KNIGHT  # Get knight movement
    directions
108
109
110
111         for direction in move_pattern:
112
113             row, column = king_field[0], king_field[1]  # Starting position
114
115
116
117             # Continue moving in the direction until blocked
118
119             if in_bound(row + direction[0], column + direction[1]):
120
121
122
123                 row += direction[0]  # Vertical movement
124
125                 column += direction[1]  # Horizontal movement
126
127
128
129                 if chessboard[row][column] != 0:  # Found piece
130
131                     if is_enemy(is_white, chessboard[row][column]) and
    chessboard[row][column].upper() == "N": # Enemy knight -> Checkmate
132
133                         return True
134
135
136
137
138         #
139         # Check for pawns
140         #
141
142
143
144
145         if is_white:
146
147             front_row = king_field[0] - 1
148
```

```
            column = king_field[1]

            enemy_pawn = "p"

        else:

            front_row = king_field[0] + 1

            column = king_field[1]

            enemy_pawn = "P"



            for direction in [1, -1]:


                if in_bound(front_row, column + direction) and chessboard[front_row][column + direction] == enemy_pawn:

                    return True



        #
        # Check for king
        #


        king_row = king_field[0]

        king_column = king_field[1]


        enemy_king = "k" if is_white else "K"



        for vertical_direction in [1, 0, -1]:


            for horizontal_direction in [1, 0 , -1]:


                if in_bound(king_row + vertical_direction, king_column + horizontal_direction) and chessboard[king_row + vertical_direction][king_column + horizontal_direction] == enemy_king:

                    return True
```

```
203
204
205
206
207
208         return False # No Checkmate found
```

Die Berechnung eines zum Vergleich gewählten Zugs braucht mit dieser Änderung in etwa 2 Sekunden (640 rekursive Minimax Aufrufe).

Zum Vergleich, ohne diese Änderung braucht der selbe Zug in etwa 3 Sekunden.
Es wurde also eine deutliche Verbesserung erreicht mit dieser einfachen Verbesserung.

8.2 `calc_piece_value_with_psqt`

Aktuell wird `calc_piece_value_with_psqt` in einer eigenen Schachbrett-Iteration berechnet, anschließend wird das Schachbrett in der Evaluierungsfunktion erneut durchlaufen um die eigentliche, erweiterte Evaluierung durchzuführen.
Stattdessen können wir einfach die `calc_piece_value_with_psqt` in der selben Iteration mit berechnen.
Bei der Validierung von wenigen Zügen macht diese Änderung keinen großen Unterschied, deswegen habe ich mich hier auf einen Vergleichswert von 1646 rekursive `MINIMAX` Aufrufen entschieden.

- Mit Verbesserung $\approx 7,5$ Sekunden.
- Ohne Verbesserung $\approx 7,5$ Sekunden.

Da der lookup nach den PSQT Werten vermutlich so schnell geht ($\mathcal{O}(1)$) , scheint selbst bei etwas breiteren Suchen diese Berechnung nicht ins Gewicht zu fallen.
Der Code ist durch diese Verbesserung dennoch verständlicher, wartbarer und evaluiert nun das Schachbrett in einer einzige Iteration.

Bekannte Probleme / Bug fixes

- ◯ Bei erzwungenem Matt für Engine gibt Backend für Engine teilweise keinen validen Zug zurück
- ◯ Wenn 3VAL verliert wird der Roboter teilweise nicht richtig aktualisiert
- ◯ Remis durch Zugwiederholung

Bug Fixes

Engine setzt dich nicht Schachmatt im Endgame.

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

Lösung: Gebe bzw. ziehe Punkte ab für früheres Matt.
Problem Identifizierung + Lösung: 4 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`**:** Optimiere 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 Angreifern).

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.

- **Tests**
  - Bennen Tests entsprechend, aktuell `test_case_1, test_case_2` ...

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.

- **Robustere Fehlerbehandlung:** Implementiere umfassendere `try...catch`-Blöcke im Frontend und Backend, um Fehler abzufangen und angemessen zu reagieren. Gib informative Fehlermeldungen an den Benutzer/die Anwendung zurück.

- **Validierung von Eingabedaten:** Validiere alle Benutzereingaben und Daten, die von externen Quellen (z.B. API-Anfragen) empfangen werden, um unerwartetes Verhalten zu verhindern.

Lesbarkeit & Wartbarkeit

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

- **API-Dokumentation (Docstrings):** Verwende Docstrings in deinen Python-Funktionen und -Klassen, um deren Zweck, Parameter und Rückgabewerte klar zu dokumentieren.
- **Refactoring komplexer Funktionen:** Teile lange und komplexe Funktionen in kleinere, besser verständliche und testbare Unterfunktionen auf.
- **Zentralisierung von Konfigurationen:** Lagere API-Endpunkte, wichtige Konstanten und Konfigurationsparameter in separate Dateien aus, um die Wartung zu erleichtern.
- **Konsistenter Code-Stil:** Achte auf eine einheitliche Formatierung und die Einhaltung von Stilrichtlinien (z.B. mit Lintern wie ESLint und Black).
- **Kommentiere komplexere Logik detaillierter:** Gehe tiefer in die Erklärungen von algorithmischen Schritten und komplexen Entscheidungsfindungsprozessen ein.

Diese Verbesserungsvorschläge sind teilweise KI generiert