Dokumentation für eine Schach-Spiel in C++

Denis Shegay
Matrikelnummer 018369052

Algorithmen und Datenstrukturen in C/C++
Bei Prof. Dr. rer. nat. Peter Gerwinski

20. Mai 2025

Verzeichnis

1 Einleitung

1.1 Motivation

Schach ist eines der ältesten Strategiespiele, das Logik, Planung und gründliche Analyse erfordert. Ich interessiere mich seit meiner Kindheit für Schach, obwohl ich schlecht spiele, aber es war diese Begeisterung, die mich dazu inspirierte, dieses Projekt umzusetzen. Durch die Kombination von Schach und Programmierung konnte ich ein Projekt erstellen, in dem Algorithmen und Datenstrukturen in der Praxis angewendet werden können.

1.2 Ziel des Projekts

Ziel des Projekts ist die Implementierung eines Konsolenschachspiels mit einem Computergegner, das Datenstrukturen, Bäume, Rekursion, Komplexitätsschätzung und Suchalgorithmen verwendet. Das Programm muss die korrekte Ausführung der grundlegenden Schachregeln, Schachmatt sowie eine strategiebasierte Zugauswahl sicherstellen.

1.3 Aufbau der Arbeit

Diese Dokumentation gliedert sich in fünf Hauptkapitel. Nach dem einleitenden Überblick werden im Kapitel 2 die technischen und konzeptionellen Grundlagen beschrieben. Kapitel 3 erläutert die Struktur und das Design des Systems. Kapitel 4 stellt die konkrete Umsetzung dar. Kapitel 5 enthält ein Fazit sowie mögliche Erweiterungen für die Zukunft.

1.4 Überblick über den Projektablauf

- Entwerfen einer Datenstruktur zur Darstellung des Bretts, der Figuren und der Züge
- Implementierung der Generierung aller möglichen Züge unter Berücksichtigung der Regeln
- Implementierung einer Prüffunktion und Filterung ungültiger Züge
- Konstruktion und Implementierung des Minimax-Algorithmus mit Alpha-Beta-Pruning
- Testen in verschiedenen Spielszenarien

Das Projekt ist ein einfaches Konsolenschachspiel mit einem Computer, implementiert in C++. Der Schwerpunkt liegt auf Algorithmen, Datenstrukturen und Analyse, was den Inhalten der Lehrveranstaltung „Algorithmen und Datenstrukturen in C/C++“ entspricht. Das Spiel unterstützt grundlegende Schachregeln, einen auf Minimax basierenden Gegnerautomaten mit Alpha-Beta-Beschneidung und eine Visualisierung des

Brettzustands über eine Textschnittstelle. Bei dem Projekt handelt es sich lediglich um ein einfaches Schachspiel, bei dem Ihr Gegner auf dem Niveau eines Schachanfängers spielt. Eine ungefähre Einschätzung seines Niveaus durch Tests liegt bei 400 ELO und darunter.

2 Technische Grundlagen

2.1 Kurzbeschreibung des Programms

Das Programm wird in C++17 mithilfe der Entwicklungsumgebung Microsoft Visual Studio unter dem Windows-Betriebssystem implementiert. Es wird nur die Standardbibliothek verwendet, was plattformübergreifende Kompatibilität und einfache Kompilierung gewährleistet. Das gesamte Projekt besteht aus einer einzigen Quelldatei und erfordert keine Abhängigkeiten oder Frameworks von Drittanbietern. und verwendet nur die Standardbibliothek, was plattformübergreifende Kompatibilität und einfache Kompilierung gewährleistet. Das gesamte Projekt besteht aus einer einzigen Quelldatei und erfordert keine Abhängigkeiten oder Frameworks von Drittanbietern.

2.2 Spielfeld und Spielfiguren

Das Spielfeld wird als zweidimensionales Array mit einer festen Größe von 8x8 dargestellt. Jede Zelle enthält ein Objekt der Struktur Piece, welches Typ und Farbe der Figur beschreibt. **Folgende Typen sind implementiert**:

- Leeres Feld
- Bauer (PAWN)
- Turm (ROOK)
- Springer (KNIGHT)
- Läufer (BISHOP)
- Dame (QUEEN)
- König (KING)

Farben sind in WHITE, BLACK und NONE eingeteilt.

2.3 Bewegungslogik

Die Bewegungsmuster der Figuren folgen den klassischen Schachregeln:

- Bauern bewegen sich vorwärts, schlagen diagonal und können sich beim Erreichen der gegnerischen Grundreihe in eine Dame umwandeln.

- Türme, Läufer und Damen bewegen sich in Linienrichtung entsprechend ihren jeweiligen Regeln.
- Springer bewegen sich in "L-Form".
- Der König kann sich ein Feld in jede Richtung bewegen.

Alle Züge werden auf ihre Gültigkeit geprüft, wobei auch geprüft wird, ob der eigene König danach im Schach steht.

2.4 Bewertung von Stellungen

Jede Stellung wird anhand eines einfachen Materialbewertungsschemas beurteilt:

- Bauer: 10 Punkte
- Springer/Läufer: 30 Punkte
- Turm: 50 Punkte
- Dame: 90 Punkte
- König: 900 Punkte

Weiße Figuren tragen positiv zum Ergebnis bei, schwarze negativ. Das Ziel des Bewertungssystems ist es, eine numerische Einschätzung der Spielsituation zu erhalten.

2.5 Zugbaumstruktur

Für die Berechnung des besten nächsten Zugs wird ein Zugbaum aufgebaut. Dabei wird jeder mögliche Zug simuliert und rekursiv weiterverfolgt. Jeder Knoten im Baum ist ein Objekt der Struktur TreeNode, welches das Spielfeld, den Zug und mögliche Folgezüge speichert. Die Suche erfolgt mittels Minimax-Algorithmus mit Alpha-Beta-Pruning, um unnötige Berechnungen zu vermeiden.

3 Konzeption

3.1 Datenstrukturen

Die Datenstrukturen sind das Fundament der gesamten Spielmechanik. Sie definieren, wie Figuren, Züge und das Brett dargestellt werden. Die Implementierung ist kompakt, aber äußerst ausdrucksstark.

Die Enumeration **PieceType** listet alle möglichen Figurentypen auf. Der Wert EMPTY steht für ein leeres Feld. Diese Definition ermöglicht es, jeden Spielstein eindeutig zu klassifizieren und erleichtert die spätere Implementierung der Bewegungslogik.

```
enum PieceType { EMPTY, PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING };
```

Listing 1: enum PieceType

Die Enumeration **Color** listet alle möglichen Figurentypen auf. Der Wert EMPTY steht für ein leeres Feld. Diese Definition ermöglicht es, jeden Spielstein eindeutig zu klassifizieren und erleichtert die spätere Implementierung der Bewegungslogik:

```
enum Color { NONE, WHITE, BLACK };
```

Listing 2: enum Color

Die Farbe einer Figur wird mit dieser Enumeration dargestellt. NONE dient zur Kennzeichnung leerer Felder oder neutraler Zustände. Diese Trennung ist essenziell, um Spielregeln wie das Schlagen von gegnerischen Figuren korrekt umzusetzen.

Die Struktur **Piece** kombiniert Typ und Farbe einer Figur. Die Methode symbol() erzeugt eine Darstellung für das Terminal. Eine Besonderheit ist, dass schwarze Figuren in Kleinbuchstaben ausgegeben werden, was eine schnelle visuelle Unterscheidung ermöglicht. Diese symbolische Darstellung ist nicht nur hilfreich für das Debugging, sondern auch für die Nutzerfreundlichkeit bei der Konsolenausgabe.

```
struct Piece {
    PieceType type;
    Color color;
    Piece(PieceType t = EMPTY, Color c = NONE) : type(t), color(c) {}
    char symbol() const {
        if (color == NONE) return '.';
        char s;
        switch (type) {
        case PAWN: s = 'P'; break;
        case ROOK: s = 'R'; break;
        case KNIGHT: s = 'N'; break;
        case BISHOP: s = 'B'; break;
        case QUEEN: s = 'Q'; break;
        case KING: s = 'K'; break;
        default: s = '.'; break;
        }
        return color == WHITE ? s : tolower(s);
    }
};
```

Listing 3: Piece

Die Struktur **Move** beschreibt einen vollständigen Spielzug, inklusive Start- und Zielkoordinaten. Die Überladung des Vergleichsoperators == ist insbesondere beim Validieren von Benutzereingaben nützlich: So kann überprüft werden, ob ein eingegebener Zug in der Liste der legalen Züge vorhanden ist.

```
struct Move {
    int fromX, fromY, toX, toY;
    bool operator==(const Move& m) const {
        return fromX == m.fromX && fromY == m.fromY && toX == m.toX &&
    toY == m.toY;
    }
};
```

Listing 4: Move

Das Schachbrett ist als 2D-Array von Piece-Objekten realisiert. Diese Entscheidung erlaubt einen schnellen Zugriff und einfache Iteration über das Spielfeld. Die Verwendung eines statischen Arrays anstelle dynamischer Container wie vector verbessert außerdem die Performance

```
typedef array<array<Piece, BOARD_SIZE>, BOARD_SIZE> Board;
```

Listing 5: typedef Board

Das Schachbrett ist als 2D-Array von Piece-Objekten realisiert. Diese Entscheidung erlaubt einen schnellen Zugriff und einfache Iteration über das Spielfeld. Die Verwendung eines statischen Arrays anstelle dynamischer Container wie vector verbessert außerdem die Performance.

Die Struktur **TreeNode** wird verwendet, um einen Entscheidungsbaum (Zugbaum) aufzubauen. Jeder Knoten enthält den aktuellen Spielstand, den letzten ausgeführten Zug und eine Liste von Nachfolgeknoten. Dies ist die Grundlage für die Minimax-Suche mit Alpha-Beta-Pruning. Die manuelle Speicherverwaltung über den Destruktor vermeidet Speicherlecks bei rekursiven Baumstrukturen.

```
struct TreeNode {
    Board board;
    Move move;
    vector<TreeNode*> children;
    TreeNode(const Board& b, const Move& m) : board(b), move(m) {}
    ~TreeNode() {
        for (auto child : children)
            delete child;
```

```
    }
};
```

Listing 6: TreeNode

3.2 Spiellogik

Die Spiellogik bildet das Herzstück der Schachumgebung. Sie entscheidet, welche Züge erlaubt sind, ob ein Spieler im Schach steht und ob ein Zug die Spielsituation verändert. Der zentrale Bestandteil dieser Logik ist die Funktion **generateLegalMoves**, welche die gültigen Züge für eine bestimmte Spielfarbe zurückgibt.

Die kleine Hilfsfunktion in **Listing 7** prüft, ob sich ein Koordinatenpaar innerhalb des 8×8-Schachbretts befindet. Sie ist essentiell, um Array-Zugriffsfehler zu vermeiden und wird in fast allen Bewegungsfunktionen verwendet.

```
bool isInside(int x, int y) {
    return x >= 0 && x < BOARD_SIZE && y >= 0 && y < BOARD_SIZE;
}
```

Listing 7: isInside

Die Funktion **generateLegalMoves** generiert alle möglichen Züge für die angegebene Farbe. Der Ablauf erfolgt wie folgt:

- **Iteration über alle Felder**: Für jede Figur der aktuellen Farbe wird geprüft, welche Züge möglich sind.
- **Typbasierte Logik**: Je nach Figurentyp werden unterschiedliche Bewegungsmuster angewendet.
- **Filterung nach Schach**: Falls **filterCheck** = **true**, werden alle Züge entfernt, die den eigenen König ins Schach setzen würden.

Block **Bauernzüge** prüft den einfachen Vorwärtszug des Bauern. Besonders interessant ist hier die Nutzung der Variable dir, die je nach Farbe die Bewegungsrichtung bestimmt eine elegante Lösung zur Vermeidung von doppeltem Code. Zudem werden Anfangszüge (zwei Felder weit), Schlagzüge und Umwandlungen behandelt.

```
if (p.type == PAWN) {
    int dir = (turn == WHITE ? -1 : 1);
    int ny = y + dir;
```

```
    if (isInside(x, ny) && b[ny][x].type == EMPTY)
        moves.push_back({ x, y, x, ny });
}
```

Listing 8: Bauernzüge

Springerzüge wird das charakteristische "LMuster des Springers überprüft. Besonders erwähnenswert ist die elegante Definition der möglichen Richtungsvektoren in einem Array.

```
if (p.type == PAWN) {
    int dir = (turn == WHITE ? -1 : 1);
    int ny = y + dir;
    if (isInside(x, ny) && b[ny][x].type == EMPTY)
        moves.push_back({ x, y, x, ny });
}
```

Listing 9: Springerzüge

Königszüge ist ähnlich wie der **Springer**, jedoch mit Bewegungen in alle Richtungen, aber nur ein Feld weit. Die Figuren **Läufer**, **Turm** und **Dame** verwenden Schleifen mit Richtungsvektoren. Die Schleife wird bei einer Kollision oder einem gegnerischen Stein abgebrochen.

Die Funktion **bool isInCheck** überprüft, ob der König einer bestimmten Farbe angegriffen wird. Sie sucht die Position des Königs und vergleicht sie mit allen möglichen Zügen des Gegners:

```
bool isInCheck(const Board& board, Color color) {
    int kingX = -1, kingY = -1;
    for (int y = 0; y < BOARD_SIZE; ++y)
        for (int x = 0; x < BOARD_SIZE; ++x)
            if (board[y][x].type == KING && board[y][x].color == color
    ) {
                kingX = x;
                kingY = y;
            }
    if (kingX == -1) return true;
    Color opponent = (color == WHITE ? BLACK : WHITE);
    auto opponentMoves = generateLegalMoves(board, opponent, false);
    for (const auto& move : opponentMoves)
        if (move.toX == kingX && move.toY == kingY)
            return true;
    return false;
```

```
16 }
```

Listing 10: bool isInCheck

Codeabschnitt in Listing 11 ist besonders elegant, da er keine Speziallogik benötigt — alle gegnerischen Züge sind bereits generiert. Dadurch wird die Funktion sehr kompakt und robust gehalten:

```
for (const auto& move : opponentMoves)
    if (move.toX == kingX && move.toY == kingY)
        return true;
}
```

Listing 11: Codeabschnitt

Die Spiellogik kombiniert somit präzise Geometrie des Schachbretts mit effizienter Filterung und strukturiertem Zugriff auf Spielfiguren.

3.3 Zugsimulation und Rücknahme

In dieser Sektion betrachten wir die Manipulation des Spielfelds durch gezielte Zugausführung und Rücknahme. Diese Mechanismen sind notwendig, um die Spielsituation für Entscheidungsprozesse temporär zu verändern oder zu analysieren.

void makeMove überträgt eine Figur vom Ursprungsfeld auf das Zielfeld. Spezialfall: Wenn ein Bauer die gegnerische Grundreihe erreicht, wird er automatisch in eine Dame umgewandelt. Diese Art der direkten Transformation ist eine bewusste Vereinfachung der üblichen Umwandlungsoptionen im Schach (Dame, Turm, Läufer, Springer), erhöht jedoch die Effizienz des Prototyps.

```
void makeMove(Board& board, const Move& m) {
    Piece moving = board[m.fromY][m.fromX];
    if (moving.type == PAWN && (m.toY == 0 || m.toY == 7))
        board[m.toY][m.toX] = Piece(QUEEN, moving.color);
    else
        board[m.toY][m.toX] = moving;
    board[m.fromY][m.fromX] = Piece();
}
```

Listing 12: makeMove

Ein globaler Stack speichert die Historie aller ausgeführten Züge. Dies ist notwendig, um Züge im Nachhinein wieder rückgängig machen zu können, zum Beispiel bei der

Analyse zukünftiger Spielverläufe (Minimax).

```
stack<Move> moveHistory;
```

Listing 13: moveHistory

undoMove kehrt einen Zug um. Sie stellt die Figur an ihre ursprüngliche Position zurück und setzt das Zielfeld auf das vorherige (möglicherweise geschlagene) Objekt. Zusätzlich wird der letzte Eintrag aus der Historie entfernt.

```
void undoMove(Board& board, const Move& m, const Piece& captured) {
    board[m.fromY][m.fromX] = board[m.toY][m.toX];
    board[m.toY][m.toX] = captured;
    moveHistory.pop();
}
```

Listing 14: undoMove

Board simulateMove erzeugt eine temporäre Kopie des Bretts, führt einen Zug aus und gibt die neue Position zurück. Wichtig: Das Original bleibt unverändert. Diese Methode ist essentiell für die Bewertung von hypothetischen Spielsituationen — etwa im Rahmen des Entscheidungsalgorithmus.

```
Board simulateMove(Board board, const Move& m) {
    makeMove(board, m);
    return board;
}
```

Listing 15: simulateMove

3.4 Entscheidungsalgorithmus

In diesem Abschnitt wird beschrieben, wie der automatische Gegenspieler seine Entscheidungen trifft. Das System verwendet eine Kombination aus Stellungsbewertung und dem klassischen Minimax-Algorithmus mit Alpha-Beta-Pruning.

evaluateBoard ist die Bewertungsfunktion, die eine einfache numerische Einschätzung der Brettstellung berechnet. Jeder Figurentyp hat einen zugewiesenen Wert (Materialgewicht). Weiße Figuren addieren, schwarze subtrahieren den Gesamtwert. Diese Heuristik dient als Grundlage für den Suchalgorithmus.

```
int evaluateBoard(const Board& board) {
    int score = 0;
    for (int y = 0; y < BOARD_SIZE; ++y)
        for (int x = 0; x < BOARD_SIZE; ++x) {
            const Piece& p = board[y][x];
            if (p.color == NONE) continue;
            int val = 0;
            switch (p.type) {
            case PAWN: val = 10; break;
            case KNIGHT:
            case BISHOP: val = 30; break;
            case ROOK: val = 50; break;
            case QUEEN: val = 90; break;
            case KING: val = 900; break;
            default: break;
            }
            score += (p.color == WHITE ? val : -val);
        }
    return score;
}
```

Listing 16: evaluateBoard

minimax implementiert den klassischen Minimax-Algorithmus mit Alpha-Beta-Schnitt. Es wird rekursiv eine bestimmte Tiefe durchsucht. Dabei werden Züge simuliert und die Positionen bewertet. Alpha und Beta dienen zur Effizienzsteigerung durch das Verwerfen von Ästen, die ohnehin nicht optimal sein können.

```
int minimax(Board board, int depth, bool maximizing, int alpha, int
    beta) {
    Color turn = maximizing ? WHITE : BLACK;
    auto moves = generateLegalMoves(board, turn, true);
    if (depth == 0 || moves.empty())
        return evaluateBoard(board);

    int best = maximizing ? INT_MIN : INT_MAX;
    for (const auto& move : moves) {
        Board copy = simulateMove(board, move);
        int score = minimax(copy, depth - 1, !maximizing, alpha, beta)
    ;
        if (maximizing) {
            best = max(best, score);
            alpha = max(alpha, best);
        } else {
            best = min(best, score);
            beta = min(beta, best);
        }
        if (beta <= alpha) break;
```

```
    }
    return best;
}
```

Listing 17: minimax

findBestMove ermittelt den besten Zug aus Sicht der übergebenen Farbe. Sie testet alle legalen Züge, simuliert sie und bewertet das Resultat mit minimax. Die Tiefe ist im Beispiel auf 3 begrenzt. Das Ergebnis ist der Zug mit dem besten erwarteten Bewertungswert.

```
Move findBestMove(Board& board, Color aiColor) {
    auto moves = generateLegalMoves(board, aiColor, true);
    if (moves.empty()) return { 0,0,0,0 };
    int bestScore = aiColor == WHITE ? INT_MIN : INT_MAX;
    Move bestMove = moves[0];
    for (const auto& move : moves) {
        Board copy = simulateMove(board, move);
        int score = minimax(copy, 3, aiColor == WHITE, -10000, 10000);
        if ((aiColor == WHITE && score > bestScore) || (aiColor ==
   BLACK && score < bestScore)) {
            bestScore = score;
            bestMove = move;
        }
    }
    return bestMove;
}
```

Listing 18: findBestMove

4 Prototypische Implementierung

In diesem Kapitel wird die konkrete Umsetzung des Spiels in C++ beschrieben. Es geht dabei nicht nur um die technischen Funktionen, sondern auch um den Ablauf und die Interaktion zwischen Benutzer und Spielsystem.

initBoard initialisiert das Schachbrett mit der Standardaufstellung. Die Figuren werden anhand eines Arrays order auf den ersten und letzten Reihen platziert, Bauern in der zweiten und siebten Reihe.

```
void initBoard(Board& board) {
    PieceType order[] = { ROOK, KNIGHT, BISHOP, QUEEN, KING, BISHOP,
   KNIGHT, ROOK };
```

```
    for (int y = 0; y < BOARD_SIZE; ++y)
        for (int x = 0; x < BOARD_SIZE; ++x)
            board[y][x] = Piece();
    for (int x = 0; x < BOARD_SIZE; ++x) {
        board[0][x] = Piece(order[x], BLACK);
        board[1][x] = Piece(PAWN, BLACK);
        board[6][x] = Piece(PAWN, WHITE);
        board[7][x] = Piece(order[x], WHITE);
    }
}
```

Listing 19: initBoard

printBoard dient der textbasierten Darstellung des Schachbretts. Durch die Verwendung der **symbol()**-Funktion jeder Figur werden sowohl weiße als auch schwarze Steine klar erkennbar abgebildet.

```
void printBoard(const Board& board) {
    cout << "a b c d e f g h";
    for (int y = 0; y < BOARD_SIZE; ++y) {
        cout << 8 - y << ' ';
        for (int x = 0; x < BOARD_SIZE; ++x)
            cout << board[y][x].symbol() << ' ';
        cout << 8 - y << ' ';
    }
    cout << "a b c d e f g h";
}
```

Listing 20: printBoard

void playGame(). Diese zentrale Spielfunktion steuert den gesamten Ablauf der Partie. Sie umfasst die Initialisierung des Bretts, die abwechselnden Spielzüge von Spieler und Automat, die Zugvalidierung sowie das Erkennen von Spielende (Matt oder Patt). Hier wird ein Eingabezug vom Benutzer analysiert, auf Gültigkeit geprüft und ausgeführt. Das Eingabeformat ist rein textbasiert (z.B. e2e4).

```
if (turn == WHITE) {
    cout << "Weisser Zug (z.B. e2e4): ";
    string input;
    cin >> input;
    if (input.length() != 4) {
        cout << "Ungueltiges Format.";
        continue;
    }
    Move m = { input[0] - 'a', 8 - (input[1] - '0'), input[2] - 'a', 8
    - (input[3] - '0') };
```

```
10      bool valid = false;
11      for (const auto& move : legalMoves) {
12          if (move == m) {
13              moveHistory.push(m);
14              makeMove(board, m);
15              valid = true;
16              break;
17          }
18      }
19      if (!valid) {
20          cout << "Ungueltiger Zug.";
21          continue;
22      }
23      if (isInCheck(board, BLACK)) {
24          cout << "Schwarz steht im Schach!";
25      }
26      turn = BLACK;
27  }
```

Listing 21: Ausschnitt: Benutzerzug

Der Automat berechnet den besten Zug mit findBestMove() und gibt ihn aus. Wenn der weiße König im Schach steht, wird dies ebenfalls angezeigt.

```
1  else {
2      cout << "Schwarzer Zug ...";
3      Move m = findBestMove(board, BLACK);
4      if (!(m.fromX == 0 && m.fromY == 0 && m.toX == 0 && m.toY == 0)) {
5          moveHistory.push(m);
6          makeMove(board, m);
7          cout << "Schwarz spielt: " << (char)('a' + m.fromX) << (8 - m.
       fromY) << (char)('a' + m.toX) << (8 - m.toY) << endl;
8      }
9      if (isInCheck(board, WHITE)) {
10         cout << "Weiss steht im Schach!";
11     }
12     turn = WHITE;
13 }
```

Listing 22: Ausschnitt: Automat-Zug

Die **main**-Funktion startet die Partie über **playGame()** und ist damit der Einstiegspunkt des Programms.

```
1  int main() {
2      playGame();
```

```
3       return 0;
4   }
```

Listing 23: main

5 Fazit

5.1 Zusammenfassung

In dieser Arbeit wurde ein funktionierender Schachautomat in C++ entwickelt, der in der Lage ist, gegen einen menschlichen Spieler zu spielen. Es wurden zentrale Konzepte aus der Informatik wie Datenstrukturen, Zustandsbäume, Suchalgorithmen, Bewertungsfunktionen und Benutzerinteraktion angewendet. Der Automat verwendet eine einfache Bewertung auf Basis von Materialwerten und trifft Entscheidungen durch Minimax mit Alpha-Beta-Pruning.

Zudem wurde ein vollständiger Spielablauf implementiert, einschließlich Initialisierung des Bretts, Visualisierung in der Konsole, Zugvalidierung und Erkennung von Schach, Schachmatt und Patt. Durch die Modularisierung der Spiellogik und die Trennung von Entscheidungs- und Ausführungsfunktionen ist der Code erweiterbar und gut wartbar.

5.2 Ausblick

Obwohl der Schachautomat bereits grundlegende Funktionalität bietet, sind zahlreiche Erweiterungen denkbar:

- **Bessere Bewertung**: Erweiterung der Bewertungsfunktion um Stellungsmerkmale wie Zentrumskontrolle, Mobilität oder Königssicherheit.
- **Eröffnungsspiel**: Integration von Eröffnungsbibliotheken für sinnvollere Anfangszüge.
- **Tiefe der Suche**: Anpassung der Suchtiefe an die Zeit oder Komplexität.
- **Interface**: Verbesserung der grafischen Darstellung z.B. mit SFML, Qt oder Webinterface.
- **Zug-Undo für Spieler**: Erweiterung der Rücknahmefunktion auch für menschliche Spieler.

Dieses Projekt kann als Basis für weiterführende Arbeiten in den Bereichen Spieltheorie, Algorithmenanalyse und Softwareengineering dienen.