Chess-Engines in Rust

Albert Eisfeld, Lukas Piorek

18. Juni 2025

1 Einleitung

Im Rahmen des Wahlmoduls „Programmieren mit Rust“ im 4. Semester des Informatik Studiums an der DHBW-Stuttgart wurden zwei Schach-Engines in der Programmiersprachen Rust entwickelt. Ziel dabei war es, einen Einblick in die Programmiersprache zu erhalten und für die Sprache übliche Konzepte zu nutzen. Im Folgenden wird die Entwicklung der beiden Chess Engines mit den Namen „Adam“ und „Thunfisch“ verglichen und reflektiert.

2 Architektur

Beide Schach-Engines wurden in der Programmiersprache Rust entwickelt, weisen jedoch unterschiedliche Schwerpunkte und architektonische Entscheidungen auf.

2.1 Allgemein

2.1.1 Aufbau

Beide Engines folgen einem modularen Aufbau, der die Kernlogik von der externen Kommunikation trennt. Die Interaktion mit Schach-GUIs oder anderen Programmen erfolgt über das standardisierte Universal Chess Interface (UCI), das als primäre Schnittstelle dient [1].

Die Engine **Adam** ist strukturell in einen `core`- und einen `models`-Bereich unterteilt. Der `models`-Bereich definiert die zentralen Datenstrukturen des Schachspiels. Hierzu gehören `ChessBoard`, welches den Zustand des Spielbretts repräsentiert, `ChessMove` zur Abbildung eines Zuges und `Piece` zur Darstellung der Figuren. Der `core`-Bereich enthält die Spiellogik, insbesondere die Zuggenerierung (`movegen`) und den Suchalgorithmus (`minimax`) [2].

Die Engine **Thunfisch** weist eine feine Modularisierung auf. Der Quellcode ist in spezifische Funktionsbereiche wie `communication`, `move_generator`, `search`, `types` und `debug` gegliedert. Diese Trennung ermöglicht es verschiedene Ansätze zu kombinieren und auszutauschen. Beispielsweise ist das `search`-Modul für die gesamte Suchlogik zuständig (inkl. Alpha-Beta-Suche, Quieszenzsuche etc.), während das `move_generator`-Modul ausschließlich die Erzeugung legaler Züge behandelt [2], [3].

2.1.2 Architektur

Rust ist eine beliebte Sprache für die Umsetzung von Chess Engines. Die Sprache verspricht hohe Geschwindigkeit, die für eine rechenintensive Anwendung wie eine Schach-Engine essenziell ist, und garantiert gleichzeitig Speichersicherheit ohne Garbage Collector. Ein weiteres Ziel war es, mit möglichst wenigen externen Abhängigkeiten auszukommen, um ein tiefes Verständnis für die Sprache und die zugrundeliegenden Algorithmen zu erlangen [2].

Bei der Engine **Adam** lag der Fokus auf einem soliden Einstieg in verschiedene Rust-Konzepte. Die Architektur spiegelt dies wider, indem sie intensiv von Rusts Typensystem Gebrauch macht. So wurden beispielsweise `structs` zur Definition der Spielkomponenten und `tuple structs` zur Erzeugung eigener, starker Typen verwendet. Dies erhöht die Typsicherheit und Lesbarkeit des Codes [2].

Bei **Thunfisch** stand die Leistungsfähigkeit und die Implementierung fortgeschrittener Techniken im Vordergrund. Zusätzlich wurde die `rayon`-Bibliothek für das Multithreading integriert. Diese Wahl wurde aufgrund der einfachen Integration und der hohen Effizienz bei der Parallelisierung des Suchalgorithmus getroffen. Die Architektur wurde von Beginn an so konzipiert, dass Komponenten wie die Zuggenerierung und die Suche unabhängig voneinander entwickelt und getestet werden konnten [2], [3], [4].

2.2 Systemkomponenten

Beide Engines bestehen im Kern aus den gleichen logischen Systemkomponenten, deren Implementierungstiefe sich jedoch unterscheidet.

- **UCI-Schnittstelle:** Diese Komponente ist für die gesamte Kommunikation verantwortlich. Sie empfängt Befehle wie `position startpos moves ...`, `go` oder `isready`, verarbeitet diese und sendet Antworten wie `bestmove e2e4` an die GUI zurück. In beiden Engines fungiert sie als Wrapper um die Kernlogik [1].
- **Spielbrett-Repräsentation:** Das Herzstück jeder Engine. Beide Engines verwenden Bitboards zur Darstellung des Spielbretts. Ein Bitboard ist eine 64-Bit-Ganzzahl, bei der jedes Bit einem Feld auf dem Schachbrett entspricht. Diese Datenstruktur ermöglicht extrem schnelle Operationen mittels bitweiser Logik, was insbesondere für die Zuggenerierung von Vorteil ist. Die zentrale `ChessBoard`-Struktur beider Engines kapselt mehrere Bitboards (eines für jede Figurenart und Farbe) sowie weitere Zustandsinformationen wie das Zugrecht, Rochademöglichkeiten und die en-passant-Regel [5], [6].
- **Zuggenerator:** Diese Komponente berechnet für eine gegebene Stellung alle legalen Züge.
  - **Adam** nutzt hierfür bitweise Operationen auf den Bitboards (Bitmasken), um die Angriffs- und Bewegungsrouten der Figuren zu berechnen und Züge zu generieren [5], [6].
  - **Thunfisch** implementiert eine weitaus komplexere und performantere Methode, die sogenannten „Magic Bitboards“. Diese Technik nutzt vorberechnete Tabellen und spezielle „magische“ Zahlen, um

die Züge für gleitende Figuren (Dame, Turm, Läufer) in konstanter Zeit zu ermitteln [5], [6], [7].

- **Suchalgorithmus & Evaluierung:** Dies ist das Gehirn der Engine. Es durchsucht den Baum der möglichen Züge, um die beste Fortsetzung zu finden.
  - **Adam** implementiert einen reinen Minimax-Algorithmus. Die Evaluierungsfunktion ist hierbei direkt in die Suche integriert und bewertet eine Stellung basierend auf dem Materialwert und dem den Positionen der jeweiligen Figuren [5], [8] .
  - **Thunfisch** verfügt über ein komplexes Suchsystem. Es verwendet eine iterative Tiefensuche (Iterative Deepening), die es der Engine erlaubt, die Suche jederzeit zu unterbrechen und den besten bisher gefundenen Zug zurückzugeben. Die Suche selbst wird durch Alpha-Beta-Pruning optimiert, um irrelevante Zweige des Suchbaums abzuschneiden. Eine Quieszenzsuche wird am Ende der normalen Suche angehängt, um den Horizont-Effekt bei gefährlichen Positionen zu mildern. Eine Transpositionstabelle speichert bereits analysierte Stellungen und deren Bewertungen, um redundante Berechnungen zu vermeiden [7], [8], [9].

Die Komponenten interagieren sequenziell: Der UCI-Listener empfängt einen Befehl, konfiguriert die Spielbrett-Repräsentation, initiiert den Suchalgorithmus, welcher rekursiv den Zuggenerator und die Evaluierungsfunktion aufruft, und gibt schließlich den besten gefundenen Zug über die UCI-Schnittstelle zurück.

2.3 Anforderungen

2.3.1 Funktional

- **UCI-Kompatibilität:** Die Engine muss das UCI-Protokoll für die Kommunikation implementieren.
- **Korrekte Zuggenerierung:** Die Engine muss in jeder beliebigen Stellung ausschließlich legale Züge generieren. Dies ist eine Grundvoraussetzung für die Spielfähigkeit.
- **Stellungs-Parsing:** Die Engine muss in der Lage sein, eine Spielstellung aus einer FEN-Zeichenkette (Forsyth-Edwards-Notation) korrekt aufzubauen.
- **Zugausführung:** Die Engine muss einen Zug finden und ausführen können, der den Spielregeln entspricht.

2.3.2 Nichtfunktional

- **Performanz:** Die Engine muss in der Lage sein, eine signifikante Suchtiefe innerhalb der vom UCI-Protokoll vorgegebenen Zeitlimits zu erreichen.
- **Korrektheit:** Die Implementierung der Algorithmen, insbesondere der Zuggenerierung, muss zu 100 % korrekt sein, was durch Perft-Tests (Performance-Tests zur Zählung der Zugmöglichkeiten) sichergestellt wird [10].

- **Lernziel:** Das Projekt diente als Lernübung, um tiefere Kenntnisse in Rust und dessen Low-Level Funktionalitäten zu erwerben. Im Vordergrund steht hierbei das Verwenden von Techniken und der Aufbau von Code der für Rust üblich ist und das Ablegen von Gewohnheiten aus anderen Programmiersprachen.

3 Umsetzung

3.1 Implementierung

Die Architektur wurde in beiden Projekten durch die konsequente Nutzung der von Rust bereitgestellten Sprachmittel umgesetzt. `structs` und `impl`-Blöcke wurden verwendet, um Daten und Verhalten zu kapseln (z.B. `impl ChessBoard { ... }`). `enums` dienten zur Modellierung von Zuständen und Varianten wie `Piece` (Bauer, Springer, …) oder `Color` (Weiß, Schwarz) [11].

Bei der Entwicklung von **Adam** wurden teilweise die Vorteile von Rusts Ownership- und Borrowing-Konzepte nicht genutzt. Insbesondere merkt man das bei der rekursiven minimax-Funktion. Um die Komplexität der Lebensdauern (`lifetimes`) zu umgehen, wurde ein funktionaler Ansatz gewählt: Anstatt den Zustand des Spielbretts direkt zu verändern (`mutable state`), erzeugt die Methode `with_move` bei jedem Zug eine neue, unabhängige Instanz des `ChessBoard`. Dies vereinfacht die Zustandsverwaltung erheblich, geht aber mit einem Performance-Overhead durch das Kopieren der Daten einher [11], [12], [13], [14], [15].

Die Implementierung von **Thunfisch** war von anderen Herausforderungen geprägt. Die Realisierung der Magic-Bitboard-Zuggenerierung war mit erheblichem Zeitaufwand verbunden. Sie erforderte nicht nur ein tiefes Verständnis des Konzepts, sondern auch die Entwicklung eines separaten Programms zur Generierung der optimalen „magischen“ Zahlen. Eine weitere große Hürde war die korrekte Integration der verschiedenen Komponenten des Suchalgorithmus. Das Zusammenspiel von iterativer Vertiefung, Alpha-Beta-Pruning, Quieszenzsuche und der Transpositionstabelle ist fehleranfällig. Ein kleiner Fehler, etwa bei der Verwaltung der Alpha-Beta-Grenzen oder beim Speichern und Abrufen von Hashes aus der Tabelle, kann die gesamte Suche unbrauchbar machen. Diese Probleme wurden durch rigorose, isolierte Tests der einzelnen Module und die Nutzung von Debugging-Visualisierungen gelöst [12], [13], [14], [15], [16], [17], [18], [19].

3.2 Mögliche Alternativen

- **Brett-Repräsentation:** Statt Bitboards hätte ein einfaches 64-elementiges Array (`[Square; 64]`) verwendet werden können. Diese Alternative ist deutlich einfacher zu implementieren, aber für die Zuggenerierung um Größenordnungen langsamer.
- **Programmiersprache:** C++ ist die traditionelle Sprache für Hochleistungs-Schach-Engines und wäre eine naheliegende Alternative gewesen. Sie bietet eine vergleichbare Performance, jedoch ohne die von Rust garantierten Sicherheiten bei der Speicherverwaltung [11].

4 Reflektion

Rückblickend lässt sich feststellen, dass beide Projekte wertvolle, aber unterschiedliche Lernerfahrungen boten.

Bei **Adam** liegt das Potenzial für zukünftige Verbesserungen vor allem bei der Leistungsoptimierung. Der nächste Schritt wäre die Implementierung von Alpha-Beta-Pruning, da der reine Minimax-Algorithmus deutlich ineffizienter ist. Darauf aufbauend könnten Multithreading zur Parallelisierung der Suche implementiert werden. Das Projekt war ein sehr guter, praxisnaher Einstieg in die Konzepte von Rust und eine gute Übung zur Erweiterung algorithmischer Kenntnisse, insbesondere im Bereich der bitweisen Logik und Operationen.

Die Entwicklung von **Thunfisch** war ein sehr ambitioniertes Unterfangen. Die größte Erkenntnis aus diesem Projekt ist, dass unverhältnismäßig viel Zeit in die Perfektionierung der Zuggenerierung geflossen ist. Obwohl die Implementierung von Magic Bitboards eine faszinierende algorithmische Herausforderung war, hätte eine einfachere Methode ausgereicht und mehr Zeit für die Optimierung der Suche und Evaluierung gelassen, welche oft einen größeren Einfluss auf die Spielstärke haben. Die größte Herausforderung war die Komplexität und das Debugging der fortgeschrittenen Algorithmen. Dennoch war das Projekt eine äußerst lohnende Erfahrung. Nach viel Arbeit mit High-Level-Sprachen wie Python und KI-Frameworks war die Rückkehr zur hardwarenahen Low-Level-Optimierung und komplexen Algorithmik eine willkommene und lehrreiche Abwechslung.

Bibliographie

[1] Stockfish, „UCI & Commands“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://official-stockfish.github.io/docs/stockfish-wiki/UCI-&-Commands.html

[2] H. Köpp, „Chess Engine Design“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://hkopp.github.io/2020/01/chess-engine-design

[3] M. Panda und M. Char, „A Modern Parallel Computing Chess Engine“, *arXiv preprint arXiv:2409.06477*, 2024, Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://web.mit.edu/dimitrib/www/MPC-MC_ArXiv2409.06477v1.pdf

[4] R. Developers, „The Rayon Book“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://github.com/rayon-rs/rayon/blob/master/README.md

[5] C. P. Wiki, „Bitboards“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Bitboards

[6] A. Healey, „Visualizing Chess Bitboards“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://healeycodes.com/visualizing-chess-bitboards

[7] C. P. Wiki, „Magic Bitboards“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Magic_Bitboards

[8] C. P. Wiki, „Move Generation“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Move_Generation

[9] J. Watzman, „Chess Move Generation With Magic Bitboards“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://essays.jwatzman.org/essays/chess-move-generation-with-magic-bitboards.html

[10] C. P. Wiki, „Perft“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/index.php?title=Perft&mobileaction=toggle_view_desktop

[11] R. U. Forum, „Optimizing Rust Borrowing Patterns in a Chess Engine“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://users.rust-lang.org/t/optimizing-rust-borrowing-patterns-in-a-chess-engine/127650

[12] C. P. Wiki, „Minimax“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Minimax

[13] iD Tech, „Minimax Algorithm in Chess, Checkers, & Tic-Tac-Toe“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.idtech.com/blog/minimax-algorithm-in-chess-checkers-tic-tac-toe

[14] Chessify, „Chess Engine Evaluation Explained“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://chessify.me/blog/chess-engine-evaluation

[15] R. Chess, „Piece-Square Tables“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://rustic-chess.org/evaluation/psqt.html

[16] C. P. Wiki, „Alpha-Beta“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Alpha-Beta

[17] S. Wiki, „Iterative Deepening“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: http://www.wiki.sharewiz.net/doku.php?id=chess:programming:iterative_deepening

[18] C. P. Wiki, „Quiescence Search“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Quiescence_Search
[19] C. P. Wiki, „Transposition Table“. Zugegriffen: 18. Juni 2025. [Online]. Verfügbar unter: https://www.chessprogramming.org/Transposition_Table