Ein algorithmenbasierter Schachcomputer

Staubach, Joel
Hochschule Mannheim
Fakultät für Informatik
Paul-Wittsack-Str. 10, 68163 Mannheim

Karhan, Marvin
Hochschule Mannheim
Fakultät für Informatik
Paul-Wittsack-Str. 10, 68163 Mannheim

***Zusammenfassung*—Der in diesem Paper vorgestellte Schachcomputer arbeitet mit bekannten Algorithmen und Hashverfahren und läuft webbasiert mit den Sprachen JavaScript und C++. Die Voraussetzung eines guten Schachcomputers liegt in einer effizienten Boardrepräsentation wie Bitboards, welche effiziente Änderungen an Figuren zulassen. Zur Suche des besten Zugs werden Suchalgorithmen, die auf dem Minimax-Algorithmus aufbauen, verwendet und die Suchgeschwindigkeit mithilfe von Pruning-Algorithmen, wie dem Alpha-Beta-Pruning-Algorithmus, verbessert. Die Qualität eines Zugs wird über eine Evaluationsfunktion anhand der Boardstellung bewertet. Der Schachcomputer erlangt eine 100 %-Siegesrate gegen alle Fairy-Stockfish-Schachcomputer der Stufe 1 bis 5 auf der Plattform Lichess.**

1. Einleitung

Seit Jahren beschäftigen sich Menschen mit der Automatisierung von Spielen. Auch Alan Turing hat sich mit der Fragestellung beschäftigt, ob es möglich ist eine Maschine Schach spielen zu lassen [1]. Heutzutage gibt es viele Schachcomputer, welche sogar wesentlich besser als Menschen spielen. Dabei gibt es Ansätze von maschinellem Lernen und algorithmische Methoden.

Dieses Paper behandelt die Entwicklung eines webbasierten Schachcomputers mit den Programmiersprachen C++ und JavaScript. Dabei werden bekannte Methoden und Algorithmen der Schachprogrammierung sowie deren Nutzen erläutert. Zuerst wird in Kapitel 2 über verwandte Arbeiten und besonders bekannte und erfolgreiche Schachcomputer berichtet. In Kapitel 3 werden die Grundlagen für die Schachprogrammierung erläutert. Besonders wird auf den Verwendungszweck von Bitboards als Boardrepräsentation für Schachbretter eingegangen und die Standards zur Kommunikation von Schachcomputern untereinander besprochen. Es wird erklärt, wie Suchalgorithmen, wie der Negamax-Algorithmus, den besten Zug ermitteln und wieso Alpha-Beta-Pruning ein wichtiger Algorithmus zur Verkürzung der Suchzeit ist. Kapitel 4 beschreibt den Aufbau des Gesamtsystems mit dem Framework Angular und dem Backend-System in C++. Um einen tieferen Einblick in die Verbesserung von Schachcomputern zu erhalten, werden in Kapitel 5 erweiterte Methoden, wie das Move Ordering und Transposition Tables, besprochen. Kapitel 6 behandelt die Ergebnisse, welche der Schachcomputer gegen verschiedene Level des Fairy-Stockfish-Schachcomputers erzielt.

2. Verwandte Arbeiten

Es gibt viele Schachcomputer, welche mit unterschiedlichen Methoden arbeiten. Zu den bekanntesten und besten gehört der Open Source Schachcomputer Stockfish. [2]. Dieser verwendet bekannte Methoden und Prinzipien der algorithmenbasierten Schachprogrammierung sowie zusätzlich ein neuronales Netz zum Evaluieren der besten Züge für eine Stellung. Stockfish gewann die letzten drei Top Chess Engine Championship (TCEC) Turniere [3]. Starke Konkurrenten sind Leela Chess Zero, welche auf einem Machine Learning Ansatz aufbaut und Komodo. Es gibt ebenfalls Ansätze [4] [5], Teile des normalen Bruteforces für Schachcomputer, z.B. im End Game, eines Spiels mithilfe von genetischen Algorithmen und deren Training abzulösen.

3. Grundlagen

3.1. Die Boardrepräsentation

1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1

Abbildung 1. Startaufstellung [eigene Darstellung]

Das Grundgerüst einer Schachanwendung besteht aus der Boardrepräsentation. Diese bestimmt die Position der Figuren auf dem derzeitigen Brett. Da die Figuren in jeder Funktion des Schachcomputers vermehrt benötigt werden, muss das Board die relevanten Informationen schnell und fehlerfrei übermitteln. Die intuitive Variante besteht aus der Verwendung eines Arrays, welches die Positionen und Typen der Figuren auf dem Brett darstellt [6]. Bei dieser Variante muss aber über alle Indizes iteriert werden, um alle Elemente zu erhalten. Deswegen ist die gängigste und schnellste Variante [6] die Verwendung von Bitboards. Ein Bitboard ist eine vorzeichenlose 64-Bit-Ganzzahl [6] [7]. Diese zeigt alle Positionen von Entitäten auf dem Brett an. Ein gesetztes Bit beschreibt dabei, dass ein Element

auf der Position vorhanden ist und ein nicht gesetztes Bit das nicht Vorhandensein.

Die Abbildung 1 zeigt die Ausgangsstellung im Schach. Die äquivalente Bitboardrepräsentation ist rechts daneben zu sehen. In einer Schachanwendung werden Bitboards für die jeweiligen Figuren und ihre Seite gespeichert. Dies bedeutet, dass beispielsweise alle Bauern auf dem Feld über ein Bitboard repräsentiert werden. Bitboards haben den Vorteil, dass sie sehr viele Änderungen schnell übernehmen können. Als Beispiel wird die folgende Zuweisung in C++ betrachtet:

```
uint64_t whitePieces = whitePawns | whiteBishops |
    whiteKnights | whiteRooks | whiteQueens |
    whiteKings
```

Diese Zuweisung erstellt ein Bitboard aus allen weißen Figuren. In einer Variante mit Arrays müsste über die einzelnen Figureneinträge iteriert werden.

Mit Bitshifts können Figuren auf dem Feld schnell bewegt werden. Beispielsweise lassen sich alle weißen Bauern mit folgendem Bitshift nach oben bewegen [6]:

```
whitePawns << 8 & fullBoard
```

In dieser Variante beschreibt *fullBoard* ein Bitboard, auf dem alle 64 Bits gesetzt sind. Dieses Bitboard verhindert bei bitweisem UND, dass das Bauern-Bitboard größer als 64 Bits werden kann. Bitboards helfen bei der schnellen Kalkulation von Zügen und Positionen [6].

Über die Figuren zu iterieren ist ebenfalls ein wichtiger Bestandteil von Schachcomputern, um beispielsweise das verbleibende Material auszurechnen. Der naive Ansatz besteht aus einer simplen Iteration der Länge 64 über das gesamte Bitboard. Peter Wegner [8] fand eine Möglichkeit, das Least Significant Bit (LSB) in einer Zahl zu entfernen, ohne die restliche Zahl zu verändern. Dies funktioniert mithilfe einer bitweisen Verundung der Zahl mit sich selbst, wobei die Zahl eins davon abgezogen wird. Ein Beispiel sieht wie folgt aus:

```
Gegeben sei:
x = 10011010
Es wird folgende Gleichung angewendet:
x = x & x - 1
Wenn nun die Werte eingesetzt werden gilt:
x = 10011010 & 10011001
x = 10011000
```

Mithilfe dieser Methode lässt sich ein schneller Figureniterator entwickeln, da nun eine Schleife verwendet werden kann, welche als Abbruchbedingung die eigene Zahl beinhaltet. Eine Pseudoimplementierung sieht folglich so aus:

```
while(pieces) {
    index = getLSBIndex()
    pieces = pieces & pieces - 1
    doSomething(index)
}
```

Die Methode hat den Nachteil, dass sie einen linearen Anstieg an Iterationen hat [9], welcher abhängig von der Anzahl der gesetzten Bits ist. Für große Zahlen gibt es eine Methode, deren Zählverhalten auf Vektoren basiert und somit für hohe Bitdichten geeignet ist [9]. Da in diesem Anwendungsfall die größte Anzahl an gesetzten Bits jedoch auf 64 limitiert ist, gibt es keine Notwendigkeit für diese Implementierung.

Der Index einer Figur lässt sich sehr schnell mithilfe des von Kim Wallisch entwickelten Algorithmus [10], welcher die Debuijn Sequenz [11] verwendet, ausrechnen. Bei diesem Ansatz wird ein Array mit 64 Indizes für die Positionen angelegt und mithilfe der folgenden Gleichung der Index des LSB für ein Bitboard errechnet:

```
((bb ^ (bb-1)) * 0x03f79d71b4cb0a89) >> 58
```

Diese Variante ist der klassischen Iteration überlegen, da die Komplexität O(1) beträgt im Vergleich zu O(n), wobei n die Anzahl an Schleifendurchläufen ist.

Damit eine Boardrepräsentation vollständig ist, müssen auch Angaben zu den derzeitigen Rochaderechten, der aktiven spielenden Seite (weiß oder schwarz) und des letzten En Passant-Feldes gespeichert werden.

3.2. Die Zugrepräsentation

Die letzte Darstellung, die betrachtet werden muss, ist die der Züge. Diese werden in Bits eines Integers dargestellt, damit eine bessere Performanz als mit einer vergleichbaren klassenbasierten Darstellung erzielt wird. Um einen Zug vollständig darstellen zu können, wird mindestens die vorherige Position und die neue Position benötigt. Mit einer 32-Bitzahl [6] lässt sich der Zugtyp, die Richtung beim Rochieren und das En Passant-Feld angeben.

Im Falle dieses Schachcomputers wird eine 16-Bitzahl verwendet, um die wichtigsten Attribute und einige hilfreiche Daten mitzugeben. Dabei bilden die ersten sechs Bits die vorherige Position ab. Mithilfe von sechs Bits lassen sich alle 64 möglichen Positionen auf dem Schachbrett darstellen. Die darauffolgenden sechs Bits beziehen sich auf die Zielposition. Um die Art des Zugs zu definieren, werden zwei Bits verwendet. Mögliche Zugtypen sind En Passant, Rochade, Figurenumwandlung und ein normaler Zug. Die letzten zwei Bits werden verwendet, um alle Figurenumwandlungen abzudecken. Ein Bauer kann sich zum Figurentyp Dame, Läufer, Springer oder Turm entwickeln. In dieser Variante werden keine En Passant-Felder und Rochadenrichtungen gespeichert, auch wenn sie die Performanz des Systems verbessern können [6].

3.3. Die FEN Notation und UCI

Um Informationen zu debuggen und um mit anderen Computern kommunizieren zu können, gibt es Standards wie die Forsyth-Edwards-Notation (FEN) und das Universal Chess Interface (UCI). Die FEN ist eine von David Forsyth entwickelte Notation und wird verwendet, um Schachpositionen mit einer Zeichenkette darzustellen [12]. Sie ist eine für den Menschen lesbare Darstellung. In der von uns entwickelten Schachanwendung wird diese Notation verwendet, um aus beliebigen Schachpositionen ein Spiel zu initialisieren. Eine FEN-Zeichenkette besteht aus sechs Teilen:

- Stellung
- Zugrecht
- Rochademöglichkeiten
- En Passant-Feld
- Halbzügen seit dem letzten Schlagen oder der letzten Bauernbewegung

- Zugnummer

Die Stellung wird reihenweise, mit schwarzer Seite beginnend, von links nach rechts dargestellt. Neue Reihen werden mit einem / eingeleitet. Die Figuren erhalten den Anfangsbuchstaben ihrer englischen Bezeichnung, beispielsweise die Dame (Queen) wird mit Q gekennzeichnet. Die einzige Ausnahme betrifft den Springer (Knight), welcher aufgrund der Namenskollision mit dem König (King) als Bezeichner N hat. Weiße Figuren werden mit Großbuchstaben bezeichnet und schwarze Figuren mit Kleinbuchstaben. Leere Felder erhalten eine Nummer von eins bis acht, welche die Anzahl an leeren Feldern in der Reihe angibt. Die FEN-Zeichenkette: *3q2k1/8/8/1Br2b2/R2N4/5r2/8/3Q2K1 w - - 0 1* sieht auf einem Brett somit wie folgt aus:

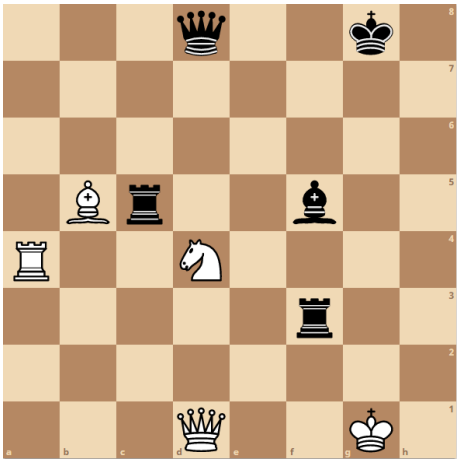

Abbildung 2. Schachposition mit FEN [eigene Darstellung]

In Abbildung 2 sind die Figuren anhand der vorherigen FEN dargestellt. Die vierte Reihe *1Br2b2* beschreibt, dass das erste Feld frei ist und darauf ein weißer Läufer (B) gefolgt von einem schwarzen Turm (r) erscheint. Danach kommen zwei freie Felder gefolgt von einem schwarzen Läufer (b) und zwei freien Feldern.

Nach der Stellungsangabe beschreibt die FEN, welche Seite am Zug ist. Dieses Attribut ist entweder weiß (w) oder schwarz (b). Die dritte Information beschreibt die Rochaderechte. Diese werden anhand der Richtung in die der König rochieren darf angegeben. Die Möglichkeiten sind für weiß Q (auf der Damenseite rochieren) und K (auf der Königsseite rochieren). Wenn beide Farben auf beiden Seiten rochieren dürfen, wird *KQkq* geschrieben. Das fünfte Attribut beschreibt, ob es ein En Passant-Feld gibt, wenn ein Bauer zwei nach vorne gelaufen ist. Ein möglicher Wert ist *c3*. Das vorletzte Attribut zählt die Anzahl an Halbzügen, die seit dem letzten Schlagen oder seit dem letzten Bauernzug vergangen sind. Dies ist wichtig, um die 50-Züge-Remisregel zu überwachen. Das letzte Attribut gibt die Anzahl an gespielten Vollzügen an, also weiß und schwarz.

Eine weitere wichtige Schnittstelle ist das UCI. Das UCI-Protokoll [13] definiert, welche Funktionen und Schnittstellen Schachcomputer anbieten müssen, um beispielsweise miteinander kommunizieren und spielen zu können. Die Ein- und Ausgabe funktioniert über die Kommandozeile. Mithilfe dieses Standards ist es möglich, auf Online-Schach-Plattformen wie *Lichess* eigene Computer zu registrieren und automatisiert spielen zu lassen. Eine Anwendung muss nicht alle UCI-Standards implementieren. Zu den wichtigsten Funktionen gehört das Erstellen eines neuen Boards (ucinewgame) sowie das Berechnen und Ausgeben des nächsten Zugs (go).

3.4. Die Zuggenerierung

Um den besten Zug zu finden und ein Schachspiel korrekt spielen zu können, muss der Computer in der Lage sein, alle möglichen Züge für eine Position auf dem Brett berechnen zu können. Dieser Schachcomputer unterscheidet dabei zwischen pseudolegalen und legalen Zügen. Unter pseudolegalen Zügen werden alle Züge verstanden, welche anhand der gegebenen Figuren und Positionen möglich sind. Legal sind diese Züge dann, wenn sie nach dem Ausführen des Zuges nicht in einem illegalen Zustand sind. Solche Fälle treten dann auf, wenn eine Figur sich nicht bewegen darf, da sie sonst den König in Schach lässt. Solche Figuren werden als gepinnte Figuren bezeichnet. Ein anderes Beispiel für einen pseudolegalen, aber nicht legalen Zug ist eine Rochade, bei der ein Feld bedroht wird, welches der König beim Rochieren übertreten würde. Um die Berechnung der einzelnen Züge zu vereinfachen, werden diese in verschiedene Arten eingeteilt und die Züge als Bitboards gespeichert.

Springer- und Königszüge sowie Bauernangriffe sind besonders einfach zu berechnen, da sie immer auf ein Feld gehen und nicht von feindlichen Figuren blockiert werden können. Dies macht sie besonders attraktiv vorzuberechnen und zu speichern, da die Züge für eine Position immer gleich bleiben.

Figuren wie Dame, Turm und Läufer sind schwieriger zu berechnen, da ihre Zugmöglichkeiten von gegnerischen und eigenen Figuren auf dem Feld blockiert werden können. Für diese Figuren gibt es den von Steffan Westcott entwickelten Kogge-Stone Algorithmus basierend auf dem Parallel Prefix Algorithmus [14] von Peter Kogge. Der folgende Pseudocode veranschaulicht dies für Züge eines Turms nach Norden:

```
north_occluded_attacks(BB bb, BB empty_bb)
{
1 bb |= empty_bb & (bb << 8);
2 empty_bb &= (empty_bb << 8);
3 bb |= empty_bb & (bb << 16);
4 empty_bb &= (empty_bb << 16);
5 bb |= empty_bb & (bb << 32);
6 return bb << 8 & FULL;
}
```

In dem gezeigten Codeausschnitt ist die Variable bb ein Bitboard mit allen vorhanden Türmen und empty_bb ein Bitboard mit allen freien Feldern auf dem Board. In der Praxis ist das die logische Negation aller Figuren auf dem Brett. Die Variable *FULL* steht hierbei für ein Bitboard auf dem alle Bits gesetzt sind. Im Folgenden wird die Veränderung des Bitboards *bb* abhängig von der Zeile im Code angezeigt:

Zeile	bb	empty_bb
0	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
	0 0 1 0 0 0 0 0	1 0 0 1 1 0 1 1
	0 0 0 0 0 0 0 0	0 1 1 0 1 1 1 1
	0 0 0 0 0 1 0 0	1 1 1 1 1 0 1 1
	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
2	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1
	0 0 1 0 0 0 0 0	1 0 0 1 1 0 1 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 1 1
	0 0 0 0 0 1 0 0	0 1 1 0 1 0 1 1
	0 0 0 0 0 1 0 0	1 1 1 1 1 0 1 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
4	0 0 1 0 0 0 0 0	1 0 0 0 1 0 0 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 1 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 1 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 1 1
	0 0 0 0 0 1 0 0	0 1 1 0 1 0 0 1
	0 0 0 0 0 1 0 0	1 1 1 0 1 0 0 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
6	0 0 1 0 0 0 0 0	0 0 0 0 1 0 0 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 0 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 0 1
	0 0 1 0 0 0 0 0	0 0 0 0 1 0 0 1
	0 0 0 0 0 1 0 0	0 1 1 0 1 0 0 1
	0 0 0 0 0 1 0 0	1 1 1 0 1 0 0 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1
	0 0 0 0 0 0 0 0	1 1 1 0 1 1 0 1

Wie zu sehen ist, lassen sich hiermit automatisch mehrere schiebbare Figuren gleichzeitig berechnen. Für Läufer ist dieses Vorhaben äquivalent und benötigt lediglich andere Bitshifts. Da die Dame nur eine Kombination aus Turm und Läufer ist, gilt dies auch für sie. Die Vorberechnung solcher Züge ist komplexer, da diese abhängig von den blockierenden Figuren auf dem Feld sind.

In der Zugberechnung gibt es noch einige Ausnahmefälle, wie En Passant-Felder für Bauern und das Schlagen von Figuren mit Bauern und dem König. Bauern sind die einzigen Figuren, deren Laufzüge unterschiedlich zu ihren Schlagzügen sind. Bei Königen muss mitbedacht werden, dass diese keine Figuren schlagen können, wenn das Feld von einer gegnerischen Figur gedeckt wird. Außerdem müssen die Züge anhand der Rochaderechte überprüft werden. Sobald sich ein Turm auf einer Seite bewegt, kann nicht mehr auf diese Seite rochiert werden. Sollte sich der König bewegen, kann überhaupt nicht mehr rochiert werden.

3.5. Die Evaluation

Die Evaluation bewertet die derzeitige Boardstellung für den aktiven Spieler. Sie ist ausschlaggebend, um den besten Zug für eine Position zu finden. Wichtig für die Suche ist, dass eine gute Evaluation scheinbar ähnliche Positionen unterschiedlich bewertet.

Ein Schachspiel besteht aus drei Phasen: der Eröffnung, der mittleren Spielphase und der Endspielphase [1, S. 8]. Für die Eröffnungsphase werden häufig Eröffnungsdatenbanken verwendet. Diese beinhalten eine große Menge an bekannten Eröffnungen und den besten darauffolgenden Zügen. Solche Eröffnungsdatenbanken können anhand von Spielen extrahiert und erstellt werden. Zu Beginn dieses Schachcomputers wurden die ersten zehn Züge von Spielen der letzten 10 Weltmeisterschaften und einiger anderer Turniere mit einer höheren Elo-Wertung als 2700 herangezogen. Dies bildetete die Eröffnungsdatenbank ab. Mit der Zeit nahm die Notwendigkeit dieser aufgrund einer besseren Evaluationsfunktion jedoch ab. Außerdem hat der Schachcomputer kein Verständnis für spezifische Eröffnungen und kann nicht gut aus diesen weiterspielen.

Die für diesen Schachcomputer verwendete Evaluationsfunktion ist minimalistisch. Sie verwendet die notwendigen Bewertungskriterien [15], wie eine Bewertung für jede Figur und deren Beweglichkeit abhängig von der Seite, die derzeit am Zug ist. In Pseudocode sieht das für die Materialbewertung wie folgt aus:

```
score = (100 * (whitePawns - blackPawns) + 320 * (
    whiteKnights - blackKnights) + 330 * (
    whiteBishops - blackBishops) + 500 * (whiteRooks
     - blackRooks) + 900 * (whiteQueens -
    blackQueens) + 3) * sideToMove
```

Die Angabe der Materialwertung wird in Centipawn angegeben. Diese ist Teil des UCI-Standards und außerdem ermöglicht sie eine feingranulare Unterscheidung der Wertigkeit von Figuren. *SideToMove* beschreibt die derzeitige Seite (weiß: 1, schwarz: -1). Wie zu sehen ist, ist ein Bauer 100 Centipawns wert. Springer sind 320 und Läufer 330 Centipawns wert. Türme sind mit 500 mehr als die Hälfte einer Dame mit 900 Centipawns wert. Die Materialbewertung ist in vielen Stellungen ein guter Indikator für eine höhere Gewinnwahrscheinlichkeit. Sollte jemand einen Bauern für einen Springer opfern, ist er somit 220 Centipawns im Vorteil.

Die Beweglichkeit einer Figur ist ebenfalls mit einer höheren Gewinnwahrscheinlichkeit verbunden [1, S. 123]. Berechnet wird diese anhand der möglichen Züge für einen Spieler. Eine weitere Möglichkeit ist die Berechnung von sicheren Zügen. Dies beinhaltet nur die Züge, welche die Figur auf einer Position zurücklassen auf welcher sie nicht geschlagen werden kann. Dies verhindert, dass aggressive Stellungen besser als sichere Stellungen bewertet werden.

Die Stärke von Figuren verändert sich mit der Anzahl der restlichen Figuren und der Position der Figur auf dem Brett [1, S. 127]. Um dies zu repräsentieren, können Tabellen, sogenannte *Piece Square Tables*, erstellt werden, um die Wertigkeit der Position einer Figur abhängig von der derzeitigen Spielphase darzustellen. Beispielsweise sollte ein König in der mittleren Spielphase in der Ecke geschützt von seinen Bauern sein. Gegen Ende des Spiels wird der König jedoch benötigt, um beispielsweise Bauern sicher auf die andere Seite zu transportieren.

Es können weitere Faktoren, wie die Anzahl an isolierten Bauern oder Doppelbauern [1, S. 8] sowie die Sicherheit eines Königs eine Rolle spielen. Jede weitere Berechnung in der Evaluation sorgt jedoch für eine längere Berechnungszeit, weswegen zwischen Geschwindigkeit und Genauigkeit abgewägt werden muss.

3.6. Die Suche nach dem besten Zug

Ein Schachspiel lässt sich aufgrund seiner Komplexität nicht von Anfang bis Ende berechnen. Da auf jeden Zug normalerweise mehr als ein Zug folgen kann, steigt die Anzahl an möglichen Zugkombinationen exponentiell mit der Tiefe. Aus diesem Grund werden Suchalgorithmen verwendet, welche anhand einer maximalen Tiefe den besten Zug auswählen.

Eine dieser Algorithmen ist der Minimax-Algorithmus. Dieser besteht aus zwei Funktionen, dem Maximieren des eigenen Wertes und dem Minimieren des gegnerischen Wertes [1, S. 132]. Für jede Tiefensuche wird jeder Knoten durchlaufen und der entgegengesetzte Algorithmus aufgerufen beginnend mit dem Maximieren. So kann der beste Zug auf Wurzelebene erhalten werden. Die folgende Darstellung zeigt einen beispielhaften Spielbaum:

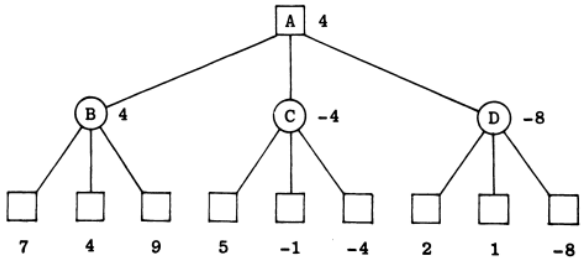

Abbildung 3. Baum mit Minimax Algorithmus [1, S. 132]

In Abbildung 3 ist zu erkennen, dass als bester Wert für *A* die Zahl vier gefunden wurde. Wenn der Minimax-Algorithmus angewendet wird, dann werden auf Wurzelebene alle Knoten (B, C, D) mit der Maximierungsfunktion aufgerufen und der Wert genommen, der am höchsten ist. Da diese Knoten keine Blattknoten sind, rufen diese für alle ihre Kinder die Minimierungsfunktion auf. Der Knoten *B* gibt dabei den Wert vier zurück. Dieser Wert ist kleiner als alle anderen Kinder von *B* und größer als die Ergebnisse der Knoten *C* und *D*. Der Grund für das wechselseitige Aufrufen von Maximierung und Minimierung liegt darin, dass davon ausgegangen wird, dass auf den eigenen besten Zug (Maximierung) auch der beste Zug des Gegners folgt. Dieser ist folglich der Zug, der für die eigene Seite am schlechtesten (minimalsten) ist. Eine kürzere Version des Minimax ist mithilfe des Negamax-Algorithmus möglich. Dieser negiert automatisch das Ergebnis des nächsten Aufrufs. Somit wird Minimierung und Maximierung mit nur einer Funktion ausgedrückt. Ein beispielhafter Pseudocode sieht wie folgt aus:

```
negaMax(depth) {
   if(depth == 0)
      return evaluate()
   else {
      max = -infinity
      for every move {
         score = -negaMax(depth - 1);
         if( score > max )
            max = score;
      }
   }
}
```

Das obige Pseudocode-Beispiel zeigt den Aufbau des Negamax-Algorithmus. Es wird eine Tiefe (depth) übergeben, um die Suche beenden zu können. Auf Blattknotenebene (Tiefe 0) wird die Evaluationsfunktion aufgerufen. Diese ist unabhängig von dem verwendeten Suchalgorithmus. Der Unterschied liegt darin, dass in der Schleife, welche den Negamax-Algorithmus rekursiv aufruft, der Wert des Ergebnisses negiert wird. Dies funktioniert, da *Maximieren = -Minimieren* gilt.

Diese und auch die Minimax-Implementierung haben jedoch zwei Nachteile. Es kann passieren, dass die Suche genau dann beendet wird, wenn eine Figur eine andere Figur geschlagen hat. Wenn diese Figur jedoch gedeckt war, ist die geschlagene Figur nicht ohne Opfer zu erhalten. In diesem Fall kann ein Zug als gut evaluiert werden, obwohl er zu einem schlechteren Ergebnis führt als vorhergesehen. Dies fällt spätestens ein paar Züge später auf, wenn die vollständige Reihenfolge von Schlägen evaluiert werden kann. Dies ist der limitierten Suchtiefe geschuldet und wird auch Horizon-Effect [16] [1, S. 146–147] genannt. Um diesen Effekt zu behandeln, gibt es die Quiescence-Suche [1, S. 146–147]. In der einfachsten Variante werden am Ende der normalen Suche alle nachträglich möglichen Schlagabtäusche durchgegangen und geschaut, ob diese einen wirklichen Vorteil bringen. Damit die Suche nicht unendlich lang verläuft, wird für jede Suche eine Evaluation durchgeführt. Sollte diese für beide Seiten zu einem stabilen Ergebnis führen, wird der Algorithmus abgebrochen.

Der zweite Nachteil der Negamax-Suche ist die Notwendigkeit der Überprüfung aller Knoten. Dies kann mit dem Alpha-Beta-Pruning-Algorithmus verhindert werden. Dieser Algorithmus gibt den gleichen Zug wie eine vollständige Suche, jedoch werden nicht alle Teilbäume untersucht [1, S. 133]. Der Algorithmus wird in Abbildung 4 veranschaulicht. Der Minimax- oder Negamax-

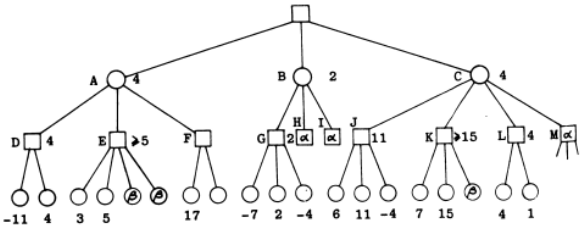

Abbildung 4. Baum mit Alpha-Beta-Pruning-Algorithmus [1, S. 133]

Algorithmus durchläuft den gesamten Teilbaum des Knotens *D* und findet den Wert vier als besten Wert. Im zweiten Teilbaum *E* wird der Wert fünf im zweiten Blatt gefunden. Da der Elternknoten von *E* (A) den minimalsten Wert sucht, können für den Knoten *E* die restlichen Blätter ignoriert werden, da fünf größer ist als vier und dementsprechend für den Knoten *E* kein besserer Zug gefunden werden kann. Dies ist ein sogenannter *Beta-Cutoff*. Für den Knoten *F* gilt dies ebenso. Ein *Alpha-Cutoff* findet dann für die Knoten *H* und *I* statt. Dies liegt daran, dass der bisherige beste Wert für den Wurzelknoten vier ist und in der Suche von *G* als bester Wert zwei gefunden wurde. Da *B* den minimalsten Wert sucht, ist der höchste Minimalwert zwei, welcher kleiner ist als vier. Somit können hier die restlichen Knoten ebenfalls ignoriert

werden. Der Alpha-Beta-Pruning-Algorithmus stellt eine große Verbesserung zur normalen Suche dar [17] [18], da viel weniger Knoten besucht werden müssen.

4. Die Architektur von NoPy++

4.1. Das Frontend mit Angular

Der entwickelte Schachcomputer besteht aus zwei Systemen. Das erste System ist die Frontend-Webanwendung. Diese wurde mithilfe des Frameworks Angular [19] entwickelt. Die Webanwendung besteht aus einer einfachen Boardanzeige und der Möglichkeit, per Drag & Drop eine Figur auf eine legale Position zu bewegen. Dabei implementiert die Anwendung einen selbst entwickelten FEN-Übersetzer. Über eine Socketverbindung baut das Frontend eine Verbindung mit dem Backend auf und bezieht für jede Positionsveränderung eine neue FEN-Repräsentation. Zusätzlich dazu erhält das Frontend die möglichen Züge für die jeweilige Seite und die derzeitige Evaluation des Schachcomputers. Dies geschieht bewusst, um keine doppelte Anwendungslogik für das Schachspiel zu implementieren. Züge können nur dann ausgeführt werden, wenn sie Teil der legalen Züge sind, die das Backend an das Frontend versendet hat. Ein auszuführender Zug wird mithilfe der Socketverbindung als standardisierter UCI-Zug ans Backend gesendet.

4.2. Das alte Backend in Python

Das ursprüngliche Backend besteht aus einem Application Programming Interface (API), welches mit dem Framework Flask [20] und einer durchgängigen Socketverbindung implementiert wurde. Diese Schnittstelle kennt zwei Anweisungen:

- das Erstellen eines neuen Boards
- das Ausführen eines Zuges

Die gesamten Schachregeln und die künstliche Intelligenz sind dort ebenfalls in Python geschrieben. Aufgrund von Performanzproblemen wurde diese Variante jedoch ersetzt und in C++ neu geschrieben. Auch die zwischenzeitliche Lösung der Nutzung von Cython für kompilierten Pythoncode erfüllte nicht die geforderten Ansprüche. Die Ausführung eines einzelnen Negamax-Aufrufs der Tiefe fünf mit der Implementierung von Alpha-Beta-Pruning dauerte drei Sekunden. Die Ausführung auf Tiefe sechs erforderte sogar 28 Sekunden.

4.3. Die neue Variante in C++

Aufgrund der mangelnden Performanz wurde das Projekt in C++ neu geschrieben. Dafür wurde auch der Socketserver in C++ mithilfe des Frameworks OATPP [21] geschrieben, um weiterhin nur zwei Systeme für das Gesamtsystem zu benötigen. Die Performanzverbesserungen sind anhand von Tabelle 1 ersichtlich. Alle Aufrufe wurden mindestens 40-mal aus der Startposition getestet und der Mittelwert errechnet.

Wie zu erkennen ist, erreicht das Umschreiben in C++ eine Performanzverbesserung von 466 % auf Tiefe 6 und auf Tiefe 7 eine Verbesserung von 1647 %.

Tabelle 1. ZEITBEDARF FÜR AUFRUFE DES NEGAMAX IM VERGLEICH

Sprache	Tiefe 5	Tiefe 6	Tiefe 7
Python (Cython)	3s	28s	84s
C++	0.01s	0.6s	5.1s

5. Algorithmen und Methoden zur Performanzsteigerung

5.1. Transposition Tables

Eine Verbesserung der Suchgeschwindigkeit kann durch Transposition Tables erzielt werden. Eine Transposition Table ist ein assoziativer Speicher, welcher durch die derzeitige Boardrepräsentation indiziert wird [1, S. 299]. Solch ein Speicher kann den letzten besten Zug speichern sowie die Alpha- und Beta-Werte der letzten Suche. Selbst wenn diese von einer niedrigeren Tiefe sind, können diese helfen *Cutoffs* in einer neuen Suche zu erzeugen, da oft der letzte beste Zug aus einer Berechnung auch in der nächsten Suche nicht schlecht ist.

Um die derzeitige Boardsituation in einen Index zu verwandeln, muss eine passende und fast perfekte Hashfunktion verwendet werden. Die Zobrist-Hashfunktion ermöglicht das einfache und schnelle Erstellen von Hashkeys für Stellungen in Spielen [22]. In einer Schachanwendung wird dabei ein pseudozufälliges Array mit 781 Einträgen generiert. Pseudozufällige Einträge können beispielsweise mithilfe eines linearen Kongruenzgenerators erstellt werden. Die pseudozufälligen Einträge versichern, dass die Ausführung immer ein konstantes Ergebnis zur Folge hat. Die 781 Einträge setzen sich aus den folgenden Eigenschaften zusammen:

- 64 x 12 = 768 (alle Figuren auf jeder Position)
- 1 (die aktive Seite)
- 4 (die vier unterschiedlichen Rochaderechte)
- 8 (die acht En Passant-Felder)

Diese Einträge werden dann abhängig von der derzeitigen Stellung mit einer *XOR*-Operation zusammengefügt. Das heißt, zur Erstellung des Boards wird jede weiße Figur mit seiner Position und jede schwarze Figur mit seiner Position per *XOR*-Operation verbunden sowie der Wert für die aktive weiße Seite hinzugefügt. Wenn sich nun etwas am Board ändert, müssen nur die Änderungen mit der *XOR*-Operation verändert werden. Wenn beispielsweise der weiße Bauer von *d2* nach *d3* zieht, muss folgende Operation geschehen, um das Board zu aktualisieren:

```
boardKey = boardKey XOR hashKeyOfwhitePawnPositionD2
    XOR hashKeyOfwhitePawnPositionD3 XOR
    hashKeyOfwhiteToMove
```

Der Zobrist-Hashalgorithmus hat bei einer 32 kb Hash Table mit 25 Bit-Keys eine Fehlerrate von 0.1 % [22], was diesen nahezu perfekt macht.

5.2. Move Ordering

Unter Move Ordering wird verstanden, die möglichen Züge anhand ihrer Wahrscheinlichkeit die besten Züge zu sein zu sortieren. Move Ordering funktioniert nur in der

Kombination mit einem Pruning-Algorithmus wie Alpha-Beta, da der Ansatz darauf aufbaut, dass frühere *Cutoffs* erzeugt werden. Wie bereits besprochen, sind die besten oder schlechtesten Züge aus Transposition Tables gute Züge zum direkten Untersuchen, da diese für *Cutoffs* sorgen oder sogar zum besten Zug führen können.

Züge, die einen Schlagabtausch gewinnen, bringen ebenfalls einen Vorteil dadurch, dass der Materialgewinn viel Einfluss auf die Bewertung hat. Um solche Schlagabtäusche schnell auszurechnen, gibt es die Static Exchange Evaluation (SEE). Die SEE berechnet einen Materialwert, der aufgrund einer Serie von Schlagabtäuschen zustande kommt [23]. Dabei werden nur die Materialwerte der Figuren betrachtet. Bei der Art der Evaluation muss darauf geachtet werden, dass es Figuren gibt (indirekte Angreifer), welche erst angreifen können, wenn die Figur, die sie blockiert, verschwindet [23]. Indirekte Angreifer können nur Figuren [23] wie Türme, Läufer oder Damen sein. Schlagabtäusche, die verloren werden, sollten generell nach hinten sortiert werden, da diese häufig ein schlechtes Ergebnis liefern und im Quiescence-Search gar nicht bewertet werden.

Es gibt noch weitere Heuristiken wie die *Killer Heuristics*, welche eine Reduzierung von bis zu 50 % an Knoten verursachen können [1, S. 134].

6. Das Ermitteln der Spielstärke

Tabelle 2. SPIELERGEBNISSE GEGEN VERSCHIEDENE STUFEN DES FAIRY-STOCKFISH-SCHACHCOMPUTERS

Stufe	1	2	3	4	5	6	7	8
Elo [1]	800	1100	1400	1700	2000	2300	2700	3000
Weiß [2]	6	6	6	6	6	6	6	6
Schwarz [3]	6	6	6	6	6	6	6	6
Gesamt [4]	12	12	12	12	12	12	12	12
Länge [5]	13	21	23	26	31	48	47	26
Siege	12	12	12	12	12	9	0	0
Remis	0	0	0	0	0	0	0	0
Siegesrate [6]	100	100	100	100	100	75	0	0

[1] die Lichess-Wertung des gegnerischen Schachcomputers
[2] die Anzahl an Spielen auf der weißen Seite (Gegner ist schwarz)
[3] die Anzahl an Spielen auf der schwarzen Seite (Gegner ist weiß)
[4] die Anzahl aller Spiele
[5] die durchschnittliche Länge aller Spiele in ganzen Zügen
[6] die Siegesrate in Prozent

Um die Spielstärke des Schachcomputers zu messen, wurde ein offizielles Bot-Konto auf der Schachseite *Lichess* erstellt und mehrere Spiele gegen verschiedene Stufen des Fairy-Stockfish-Schachcomputers ausgewertet. Der Fairy-Stockfish-Schachcomputer ist eine Abwandlung des originalen Stockfish-Schachcomputers und verwendet dessen Evaluation. Die Tabelle 2 zeigt die Ergebnisse. Die Elo-Wertung bezeichnet die Wertung des gegnerischen Schachcomputers und wurde von *Lichess* vergeben. Beispielsweise hat der Schachweltmeister Magnus Carlsen am 10.07.2021 auf der Plattform Lichess eine Blitz-Wertung von 3102 [24] und ist damit der beste Spieler in dieser Kategorie. Die Kategorie Blitz beinhaltet Spiele, die zwischen drei und acht Minuten dauern.

In diesem Testszenario wurden zwölf Spiele mit jeweils gleicher Anzahl an schwarzer und weißer Seite auf unbegrenzte Zeit gegen die Stufen des Fairy-Stockfish-Schachcomputers gespielt. Dabei wählt unser Schachcomputer immer nach fünf Sekunden einen Zug aus. Diese Zeitbeschränkung hat zur Folge, dass unser Schachcomputer im Durchschnitt sieben bis acht Züge voraussehen kann, wenn Schlagabtäusche mithilfe der Quiescence-Suche nicht mitgezählt werden.

Zu beobachten ist, dass die ersten fünf Stufen, also die mit einer Elo-Wertung von 800 bis 2000, niemals gegen unseren Schachcomputer gewinnen. Außerdem erhöht sich die durchschnittliche Länge der Spiele mit der Stärke des Gegners. Beispielsweise beträgt die durchschnittliche Spiellänge auf der ersten Stufe 13 Ganzzüge und auf der fünften Stufe 32 Ganzzüge. Auf Stufe 6 mit einer Elo-Wertung von 2300 erreicht unser Schachcomputer lediglich eine Siegesrate von 75 % und auf den darauffolgenden Stufen eine Siegesrate von 0 %. Die längsten Spiele hat unser Schachcomputer im Durchschnitt auf der Stufe 6 mit 48 Ganzzügen. Ab den darauffolgenden Stufen nimmt auch die durchschnittliche Spiellänge ab. Das Phänomen lässt sich eventuell so klären, dass die Schachcomputer deutlich bessere Züge finden und somit schneller eine Mattsituation finden können. Ähnlich verhält sich dies auch bei den ersten fünf Stufen. Diese geben beispielsweise langfristig Figuren auf und werden somit schneller von unserem Schachcomputer matt gesetzt. Somit sind auch Siegesraten von 0 % nicht unwahrscheinlich, da Schachcomputer deterministisch arbeiten. Das bedeutet, wenn die Evaluation eines Schachcomputers signifikant besser ist als die des Gegners, wird diese immer gewinnen. Wenn die Evaluationen jedoch nicht immer besser sind, entstehen Szenarien, wie in den Spielen gegen Stufe 6. So lässt sich auch erklären, dass es niemals zu einem Remis kommt. Die einzige Möglichkeit würde dann entstehen, wenn beide Schachcomputer nahezu identisch gut sind. Dies passiert beispielsweise häufig bei den besten Schachcomputern, wenn diese gegeneinander spielen.

Der aktuelle Stand der Datenlage ist, dass unser Schachcomputer zwischen einer *Lichess*-Wertung von 2300 bis 2700 liegt. Die Datenmenge ist jedoch sehr klein und muss in Zukunft durch mehr Spiele und verschiedene Gegner erweitert werden. Spiele gegen menschliche Gegner, wie Schachgroßmeister, könnten ebenfalls zu einer besseren Einschätzung unseres Schachcomputers führen.

7. Fazit und Ausblick

Die Grundlage zur Erstellung eines effizienten Schachcomputers liegt in der Verwendung von Bitboards als Boardrepräsentation. Diese erlauben schnelle Operationen auf den Feldern und Zählungen von Figuren. Zur Kommunikation zwischen Schachcomputern und grafischen Schachoberflächen gibt es Standards wie die UCI und die FEN. Bei der Generierung von möglichen Zügen helfen Kombinationen aus vorberechneten Zügen für beispielsweise Springer und Algorithmen wie der Kogge-Stone-Algorithmus für Dame, Turm und Läufer. Die Evaluation von Spielstellungen sollte anhand der verschiedenen Spielphasen geschehen. Zu den wichtigsten Bewertungskriterien gehören das Material und die Beweglichkeit der Figuren [15].

Da Schach aufgrund seiner Komplexität nicht von Anfang bis zum Ende berechenbar ist, werden Suchal-

gorithmen wie der Negamax-Algorithmus verwendet, um Züge anhand einer maximalen Tiefe berechnen zu können. Damit unwichtige Bäume nicht untersucht werden müssen, gibt es Algorithmen wie den Alpha-Beta-Pruning-Algorithmus. Methoden wie die Benutzung von Transposition Tables und Move Ordering verbessern die Geschwindigkeit der Suche weiterhin, indem weniger Teilbäume besucht werden müssen.

Die behandelten Themen bestehen aus vielen Teilen mit unterschiedlicher Komplexität. Beispielsweise gibt es für die SEE eine Most-Valuable-Victim/Least-Valuable-Attacker-Tabelle, die aufgrund der Übersichtlichkeit nicht erwähnt wurde. Dieses und weitere Verfahren sind ebenfalls Teile unseres Schachcomputers und überschreiten den Rahmen des Papers.

Der entwickelte Schachcomputer arbeitet als Webanwendung mit dem Framework Angular im Frontend und C++ im Backend. Die Züge reichen aus, um den Fairy-Stockfish-Schachcomputer von Stufe 1 bis Stufe 5 mit einer Wahrscheinlichkeit von 100 % zu besiegen. Die Spielstärke des Schachcomputers liegt ungefähr zwischen einer Lichess-Wertung von 2300 bis 2700. Der Schachcomputer verliert gegen den Fairy-Stockfish-Schachcomputer auf Level 7 und Level 8 und kann somit noch verbessert werden.

Besonders die Tiefe des Schachcomputers sollte verbessert werden, da beispielsweise der Schachcomputer Stockfish eine Tiefe von 24 Zügen erreicht, in der gleichen Zeit, in der dieser Schachcomputer eine Tiefe von acht Zügen erreicht. Eine Verbesserungsmöglichkeit ist beispielsweise die Verwendung von Magic Bitboards [23] zur Vorberechnung von Zügen für Damen, Türme und Läufer. Außerdem kann untersucht werden, wie sich die Spielstärke des Schachcomputers verändert, wenn anstelle einer statischen Evaluation ein Efficiently Updatable Neural Network (NNUE) [25] oder ein Neuronales Netz wie beispielsweise ein Convulational Neuronal Network, zur Zugauswahl verwendet wird [26].

Abkürzungen

API Application Programming Interface
FEN Forsyth-Edwards-Notation
LSB Least Significant Bit
NNUE Efficiently Updatable Neural Network
SEE Static Exchange Evaluation
TCEC Top Chess Engine Championship
UCI Universal Chess Interface

Literatur

[1] D. Levy, Hrsg., *Computer Chess Compendium*. Springer New York, 1988. DOI: 10.1007/978-1-4757-1968-0.

[2] Stockfish. (2021). „Stockfish“, Adresse: https://stockfishchess.org/ (besucht am 10. 07. 2021).

[3] Wikipedia. (2021). „Wikipedia“, Adresse: https://de.wikipedia.org/wiki/Top_Chess_Engine_Championship (besucht am 10. 07. 2021).

[4] E. Vázquez-Fernández und C. A. C. Coello, „An adaptive evolutionary algorithm based on tactical and positional chess problems to adjust the weights of a chess engine“, in *2013 IEEE Congress on Evolutionary Computation*, 2013, S. 1395–1402. DOI: 10.1109/CEC.2013.6557727.

[5] N. Lassabe, S. Sanchez, H. Luga und Y. Duthen, „Genetically Programmed Strategies for Chess Endgame“, in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, Ser. GECCO '06, Seattle, Washington, USA: Association for Computing Machinery, 2006, S. 831–838. DOI: 10.1145/1143997.1144144.

[6] B. Boskovic, S. Greiner, J. Brest und V. Zumer, „The representation of chess game“, in *27th International Conference on Information Technology Interfaces, 2005.*, 2005, S. 359–364. DOI: 10.1109/ITI.2005.1491153.

[7] P. S. Segundo, R. Galan, F. Matia, D. Rodriguez-Losada und A. Jimenez, „Efficient Search Using Bitboard Models“, in *2006 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, 2006, S. 132–138. DOI: 10.1109/ICTAI.2006.53.

[8] P. Wegner, „A Technique for Counting Ones in a Binary Computer“, *Commun. ACM*, Jg. 3, Nr. 5, S. 322, Mai 1960. DOI: 10.1145/367236.367286.

[9] S. Berkovich, G. M. Lapir und M. Mack, „A Bit-Counting Algorithm Using the Frequency Division Principle“, *Softw. Pract. Exper.*, Jg. 30, Nr. 14, S. 1531–1540, Nov. 2000. DOI: 10.1002/1097-024X(20001125)30:14%3C1531::AID-SPE347%3E3.0.CO;2-A.

[10] Chessprogramming. (2021). „BitScan“, Adresse: https://www.chessprogramming.org/BitScan (besucht am 10. 07. 2021).

[11] N. Bruijn, de, *Acknowledgement of priority to C. Flye Sainte-Marie on the counting of circular arrangements of 2^n zeros and ones that show each n-letter word exactly once*, English, Ser. EUT report. WSK, Dept. of Mathematics and Computing Science. Technische Hogeschool Eindhoven, 1975. Adresse: https://pure.tue.nl/ws/portalfiles/portal/1875614/252901.pdf (besucht am 10. 07. 2021).

[12] A. Iqbal, „An Algorithm for Automatically Updating a Forsyth-Edwards Notation String Without an Array Board Representation“, in *2020 8th International Conference on Information Technology and Multimedia (ICIMU)*, 2020, S. 271–276. DOI: 10.1109/ICIMU49871.2020.9243487.

[13] S.-M. Kahlen. (Apr. 2004). „The UCI protocol“, Adresse: http://wbec-ridderkerk.nl/html/UCIProtocol.html (besucht am 10. 07. 2021).

[14] P. M. Kogge und H. S. Stone, „A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations“, *IEEE Transactions on Computers*, Jg. C-22, Nr. 8, S. 786–793, 1973. DOI: 10.1109/TC.1973.5009159.

[15] C. E. Shannon, „XXII. Programming a computer for playing chess“, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Jg. 41, Nr. 314, S. 256–275, 1950. DOI: 10.1080/14786445008521796.

[16] H. J. Berliner, „Some Necessary Conditions for a Master Chess Program“, in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Ser. IJCAI'73, Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, S. 77–85. Adresse: https://dl.acm.org/doi/10.5555/1624775.1624786 (besucht am 10. 07. 2021).

[17] J. Pearl, „The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality“, *Commun. ACM*, Jg. 25, Nr. 8, S. 559–564, Aug. 1982. DOI: 10.1145/358589.358616.

[18] J. R. Slagle und J. E. Dixon, „Experiments With Some Programs That Search Game Trees“, *J. ACM*, Jg. 16, Nr. 2, S. 189–207, Apr. 1969. DOI: 10.1145/321510.321511.

[19] Angular. (2021). „Angular“, Adresse: https://angular.io/ (besucht am 10. 07. 2021).

[20] Flask. (2021). „Flask“, Adresse: https://flask.palletsprojects.com/en/2.0.x/ (besucht am 10. 07. 2021).

[21] OATPP. (2021). „OATPP“, Adresse: https://oatpp.io/ (besucht am 10. 07. 2021).

[22] A. Zobrist, „A New Hashing Method with Application for Game Playing“, *ICGA Journal*, Jg. 13, S. 69–73, 1990. Adresse: https://minds.wisconsin.edu/bitstream/handle/1793/57624/TR88.pdf?sequence=1&isAllowed=y (besucht am 10. 07. 2021).

[23] F. Reul, „New architectures in computer chess“, 2009. Adresse: https://www.cwi.nl/about/library/documents/229592.pdf (besucht am 10. 07. 2021).

[24] Lichess. (2021). „DrNykterstein Blitz-Statistiken“, Adresse: https://lichess.org/@/DrNykterstein/perf/blitz (besucht am 10. 07. 2021).

[25] Y. Nasu, „NNUE: Efficiently Updatable Neural-Network-basedEvaluation Functions for Computer Shogi“, *Ziosoft Computer Shogi Club*, 2018. Adresse: https://www.apply.computer-shogi.org/wcsc28/appeal/the_end_of_genesis_T.N.K.evolution_turbo_type_D/nnue.pdf (besucht am 11. 07. 2021).

[26] B. Oshri, „Predicting Moves in Chess using Convolutional Neural Networks“, 2015. Adresse: http://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pdf (besucht am 10. 07. 2021).