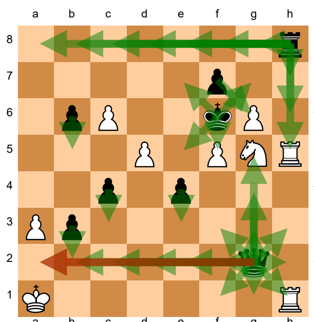
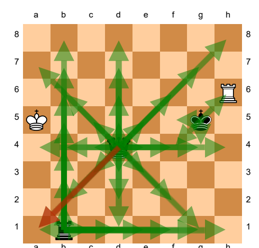
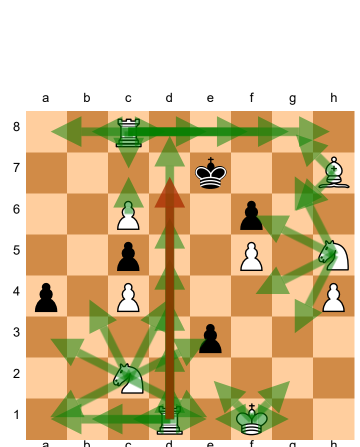
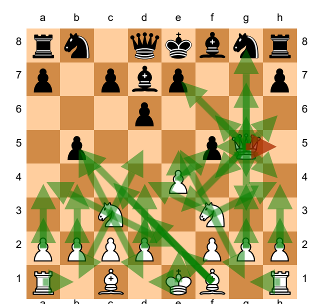
Effizienzanalyse des Minimax-Algorithmus in Bezug auf Schach

Leo Blume, 12 J., Werdener Gymnasium









Inhaltsverzeichnis

1. Einleitung
2. Schach-Engine-Verfahren
   2.1. Monte-Carlo-Verfahren
   2.2. Minimax-Verfahren
   2.3. Alpha-Beta-Pruning
3. Effizienzanalyse
   3.1. Vor- und Nachteile der beiden Methoden im Vergleich
   3.2. Effizienzanalyse Minimax
      3.2.1. Prinzip (und Problem) des exponentiellen Wachstums
      3.2.2. Warum kann es keinen perfekten Spielbaum geben?
      3.2.3. Was ist eine gute Spielbaumtiefe für den Minimax-Algorithmus im Browser?
4. Entwicklung einer eigenen Schachengine im Browser
   4.1. Implementierung Minimax-Verfahren
   4.2. Grafische Oberfläche und Design
5. Fazit
6. Quellen / Literatur

1. Einleitung

Ich spiele Schach schon seit meinem fünften Lebensjahr. Das Spiel lernte ich an einer Schach-AG an meiner früheren Grundschule. Schon bald konnte ich gut genug spielen, um einem Schachverein in der Nähe, den Schachfreunden Essen-Werden, beizutreten. Dort lernte ich immer bessere Spielarten und Strategien, sodass ich mittlerweile auf der Online-Plattform lichess.org[1] (auf der ich gerade in Corona-Zeiten sehr viel gespielt habe) eine Schnellschach-Wertung von 1750 habe.

Seit der 5. Klasse interessiere ich mich schon für das Programmieren. Am Anfang erstellte ich „einfache“ Spiele auf Scratch, aber ich fing recht schnell an, mich auch für nicht-visuelle Sprachen begeistern zu können.

Besonders interessiert war ich von Anfang an von JavaScript, der einfachen Integration in den Browser und der HTML5-Canvas-API, die das ermöglichte, was auch mit Scratch möglich war, aber es gab deutlich mehr Möglichkeiten (wie die Darstellung von dynamischen Variablen, multidimensionale Arrays, Bearbeitung von einzelnen Pixeln etc.).

Als es an der Zeit war, ein Jugend forscht-Projekt für dieses Jahr auszuwählen, entschied ich mich dazu, diese beiden Hobbys zu kombinieren und mich mit Schachcomputern bzw. Engines zu beschäftigen.

Schon vorher hatte ich in JavaScript ein Schachprogramm zum Training von Mattreflexen mit dem Namen 10000mates[2] entwickelt (es hat aber noch einige Fehler, wie u.a. eine fehlerhafte Highscorefunktion).

Aber bei diesem Projekt wollte ich mich auf eine Engine spezialisieren, die das Schachspiel einerseits so effizient wie möglich lösen konnte, und auf der anderen Seite auch nicht zu schwer umzusetzen war. Ich fragte mich am Anfang natürlich, was Top-Engines wie Stockfish[3] (open-source), AlphaZero[4][16] (Google), Komodo[5] (Chessbase) & Co. verwendeten.

Stockfish und Komodo erstellen hierbei direkt einen Zugbaum und evaluieren jede mögliche Stellung mithilfe des Alpha-Beta-Prunings (siehe 2.3); AlphaZero setzt auf ein Monte-Carlo-Verfahren[6] („Zufallswegverfahren“) (siehe 2.1), das sehr häufig nacheinander angewandt wird, weswegen auch deutlich aggressivere Varianten vom Algorithmus gespielt werden.

2. Schach-Engine-Verfahren

2.1. Monte-Carlo-Verfahren

Beim sogenannten Monte-Carlo-Verfahren wird kein vertieftes Schachwissen vorausgesetzt, d.h. mit Ausnahme der Schachregeln und der möglichen Ausgänge einer Partie ist dem Algorithmus nichts bekannt.

Es ist somit für den Algorithmus erst einmal egal, ob ein Zug eine schlechte Stellung hinterlässt oder sogar Material verliert. Weil extrem viele Spiele so bis zum Ende „zufällig“ durchgespielt werden, lernt dieser Algorithmus immer weiter dazu.

Damit der Algorithmus die maximale Spielstärke erreicht, muss er jedenfalls viele Stunden Training mit sich selbst absolvieren und diese durch künstliche Intelligenz auswerten.

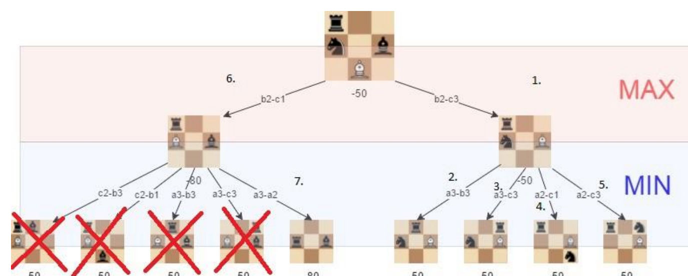
Dieses Verfahren wird unter anderem von Schachprogrammen wie AlphaZero oder LeelaZero verwendet.

2.2. Minimax-Verfahren

Das Minimax-Verfahren[7] [8] verwendet, ganz anders als das Monte-Carlo-Verfahren, einen festen Spielbaum und eine so gut wie möglich ausgeprägte Evaluationsfunktion mit so viel Schachwissen wie möglich.

Der Algorithmus wird meist mit einer durch eine Tiefe limitierte rekursive Funktion entwickelt. So wird für die Position nach jedem Zug nach einer bestimmten Stellung der Algorithmus erneut ausgeführt. Wird die Tiefe 0 erreicht, wird für jede mögliche Stellung, die nun auftreten kann, eine Evaluationsfunktion verwendet, um den Wert dieser Stellung in dieser Tiefe zu berechnen.

Wenn der Spieler, für den der beste Zug gefunden werden soll, Weiß ist, so wird der Wert am Anfang maximiert. Weil beim Zug danach logischerweise Schwarz am Zug ist, versucht dieser, den Wert zu minimieren. Dieses Diagramm zeigt es vielleicht ganz gut:



Quelle: freeCodeCamp.org

Das Beispiel ist natürlich stark auf ein 3x3-Spielfeld mit nur 4 Figuren vereinfacht, aber das Konzept bleibt dasselbe.

2.3. Alpha-Beta-Pruning

Alpha-Beta-Pruning oder einfach nur Alpha-Beta-Suche ist eine Erweiterung für den Minimax-Algorithmus, die die Effizienz dessen verbessert, indem bestimmte Züge ignoriert werden, die sicher nicht in die Endentscheidung des besten Zuges einfließen werden.

3 Effizienzanalyse

3.1 Vor- und Nachteile der beiden Verfahren im Vergleich

Vorteile des Monte-Carlo-Algorithmus:

- Nach langem Training ist er sehr stark und effizient.
- Der unvoreingenommene Spielstil sorgt dafür, dass auch Figurenopfer und Stellungsverschlechterungen in Kauf genommen werden.
- AlphaZero, der aktuell stärkste Schachalgorithmus der Welt, basiert auf diesem System.

Nachteile des Monte-Carlo-Algorithmus:

- Kompliziert zu programmieren
- Braucht enorm viel Zeit zum Trainieren oder eine große Datenbank, um einen angemessenen Stärkegrad zu erreichen
- Normale PCs und insbesondere Smartphones haben nicht genug Leistung, um diesen Algorithmus anzutreiben.

Vorteile des Minimax-Algorithmus:

- Tiefe (und somit Stärke) einfach einstellbar
- Recht leicht zu programmieren
- Braucht nicht viel CPU-Leistung
- Kann direkt („out-of-the-box") verwendet werden, ohne irgendeine Form von Training
- Hat noch weitere Einstellungen, die deutlich einfacher vorgenommen werden können
- Kann durch Alpha-Beta-Pruning deutlich stärker gemacht werden.

Nachteile des Minimax-Algorithmus:

- In der Basiskonfiguration recht langsam
- Bei geringer Tiefe schlechte Spielstärke

Da der Monte-Carlo-Algorithmus auf der einen Seite kompliziert umzusetzen ist und außerdem noch stundenlanges Training braucht, um auf Höchstleistung zu funktionieren, werde ich mich in diesem Projekt auf den Minimax-Algorithmus spezialisieren. Dieser hat auch den weiteren Vorteil, durch Erweiterungen wie das Alpha-Beta-Pruning noch effizienter und besser zu werden und hat viele Optionen, die verwendet werden können.

3.2. Effizienz-Analyse Mini-Max

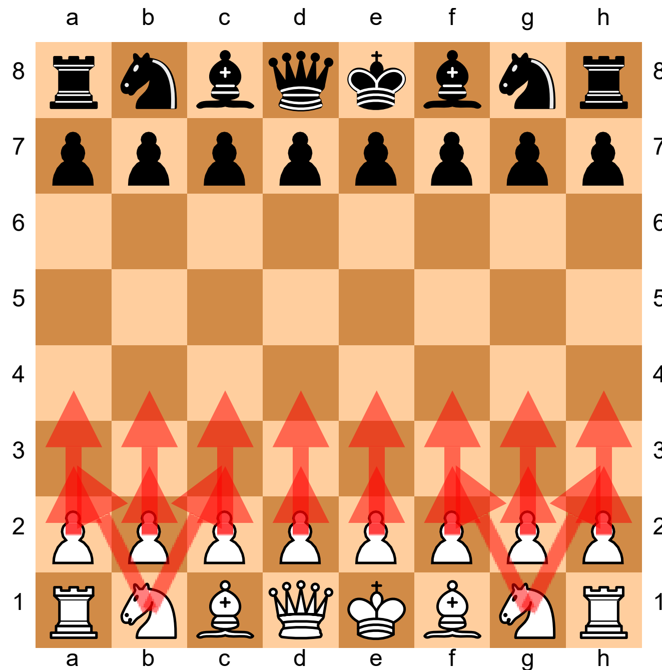
3.2.1. Prinzip (und Problem) des exponentiellen Wachstums

Eine Schachpartie beginnt immer mit derselben Stellung: der sogenannten Grundstellung. Am Anfang hat Weiß 20 Zugvarianten.

Aus diesen 20 Varianten ergeben sich danach auch 20 verschiedene Positionen, die Schwarz mit 20 verschiedenen Antwortmöglichkeiten erwidern kann. Allein nach zwei Halbzügen können demnach schon 400 nicht-identische Positionen auftreten. Und diese Zahl wächst bis ins offensive Mittelspiel immer weiter (s.u.).

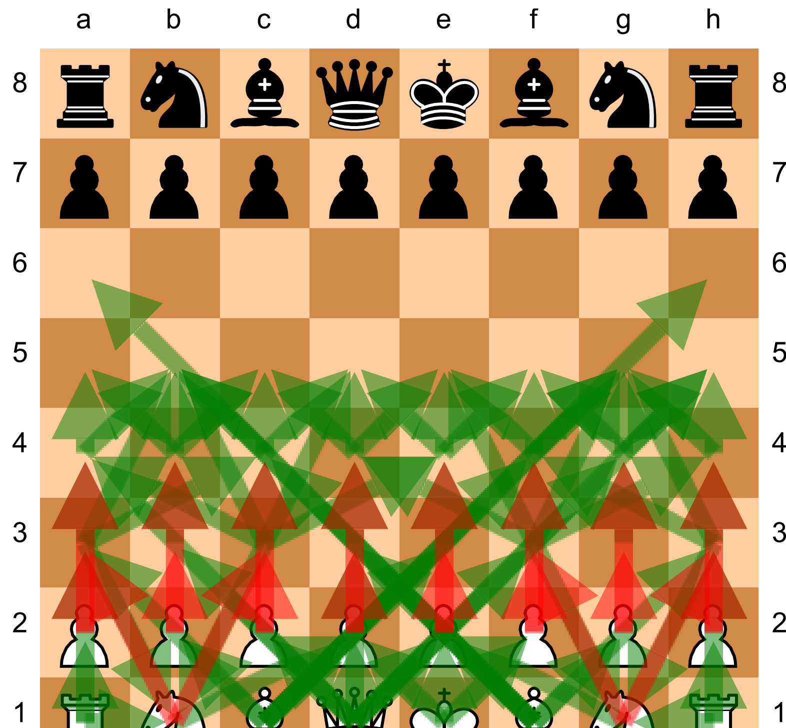
3.2.2. Warum kann es keinen perfekten Spielbaum geben?

Sehen wir uns noch einmal das Beispiel von gerade an.



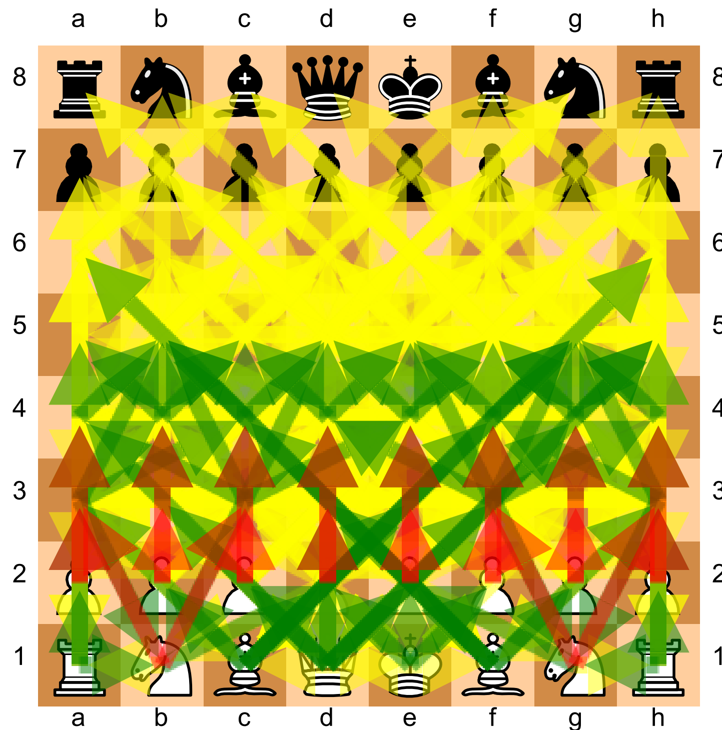
Die möglichen Anfangzüge von Weiß. Erstellt mit Python [9]und chess.py[10]. Berechnungszeit: 0.4s

Dieses Bild zeigt alle 20 möglichen Anfangszüge von Weiß. Es sind genau 20: zwei verschiedene Züge mit jedem der acht Bauern, und noch zwei mögliche Züge mit jedem der zwei Springer, die hier auf dem Brett vorhanden sind. Betrachten wir nun einmal die Züge von Weiß nach allen möglichen Zügen von Schwarz:



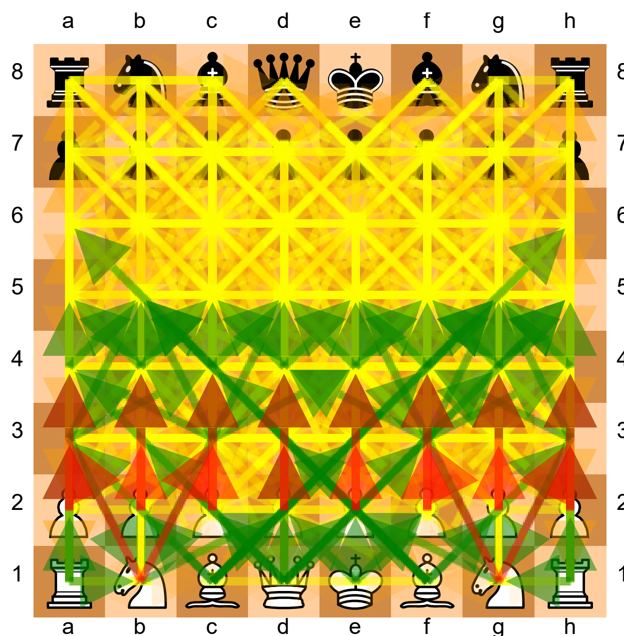
Die möglichen 2. Züge von Weiß. Erstellt mit Python und chess.py. Berechnungszeit: 1.3s

Die Anzahl der möglichen weißen Züge hat sich fast verdoppelt (20 → 38). Die Anzahl der möglichen Positionen steigt dabei natürlich noch schneller, da jedem Zug von Weiß auch noch ein Zug von Schwarz folgt, der die Position ebenfalls verändert. Die möglichen 3. Züge für Weiß sind diese:



Die möglichen 3. Züge für Weiß. Erstellt mit Python und chess.py. Berechnungszeit: 46.8s

Es ist klar zu erkennen, dass auch die Anzahl der Züge exponentiell ansteigt. Nur zum Spaß habe ich noch einmal den 4. Zug von Weiß mit meinem Programm berechnen lassen. Es hat länger gedauert, als ich dachte.



Die möglichen 4. Züge für Weiß. Erstellt mit Python und chess.py. Berechnungszeit: 34689.4s

Um dieses Bild herzustellen, hat mein PC mehr als neun Stunden rechnen müssen. Durch diese Graphik wird die schiere Anzahl der Zugmöglichkeiten aus meiner Sicht auch auf eine sehr ästhetische Weise ersichtlich.

Nach dem vierten Halbzug (dem zweiten Zug von Schwarz) gibt es bereits 197.281 verschiedene Positionen.

Aber dies sind nur Züge von Anfang an. Wenn man jede einzelne Schachspielmöglichkeit, die je eintreten könnte, zählen würde, läge man bei etwa 10^{120}. Und das ist noch eine geringe Schätzung.[11]

Zum Vergleich:

Im beobachtbaren Universum gibt es nur etwa 10^{84} bis 10^{88} Atome.[12]

Ein Schachzug braucht etwa 4 Bytes Speicherplatz, eine durchschnittliche Partie mit 40 Zügen also ungefähr 160 Bytes bzw. 1280 Bits. Ein Bit kann nach neueren Erkenntnissen in 12 Atomen gespeichert werden[13][14], demnach bräuchte man 14.360 Atome, etwa 10^4 für ein Schachspiel.

Sollte man also JEDES einzelne Atom im beobachtbaren Universum dafür verwenden, so viele Schachspiele wie möglich zu speichern, hätte man nur 0,000 000 000 000 000 000 000 000 000 000 001 % - also ein Quadrilliardstel eines Trillionstel der Spiele tatsächlich gespeichert.

3.2.3. Was ist eine gute Spielbaumtiefe für den Minimax-Algorithmus im Browser?

Auf einer Website sollte die Berechnungszeit für einen berechneten Schachzug nicht mehr als 30 Sekunden betragen. Da man beim Minimax-Algorithmus – anders als z.B. beim Monte-Carlo-Verfahren – nicht einfach den Algorithmus direkt beenden und einen Zug erzwingen kann (es sei denn, alle auf der aktuellen Tiefe entstehenden Stellungen wurden bereits evaluiert, was ein deutlich komplizierteres Programm zur Folge hätte), muss eine klar festgelegte Tiefe für jeden Zug bestimmt werden. Natürlich kann man auch den Nutzer dazu bringen, selbst eine Tiefe auszuwählen, aber es sollte definitiv eine „Empfohlen"-Einstellung geben, die den bestmöglichen Zug in weniger als 30 Sekunden zu finden. Ich habe mit JavaScript ein paar Tests gemacht, wie lang es mit verschiedenen Tiefen dauert. Das Ergebnis: Aktuell kann in der Eröffnung und im Mittelspiel eine Tiefe von 3 Halbzügen verwendet werden, im Endspiel kann es auf bis zu 6 (je nach Figurenanzahl) erhöht werden.

4. Umsetzung einer eigenen Schachengine im Browser

4.1 Implementierung Minimax-Verfahren

In JavaScript gibt es zwei Möglichkeiten, einen Minimax-Algorithmus in die Tat umzusetzen – einmal mit zwei Funktionen, einer min- und einer max-Funktion, oder auch mit einer einzigen sogenannten NegaMax-Funktion. Die NegaMax-Funktion ist zwar eigentlich effizienter, aber die zwei Funktionen sind anschaulicher und enthalten weniger Konditional-statements, sodass ich diese für das Beispiel hier verwenden werde. Die Max-Funktion nimmt einen Spieler, eine Tiefe und eine Position. Falls das Spiel vorbei oder die Tiefe 0 ist, wird die Funktion sofort beendet. Dann wird der beste Zug aller möglichen Züge mithilfe der Min-Funktion berechnet. Zurückgegeben wird der Wert dieses besten Zuges. Falls die Tiefe der gewünschten Tiefe entspricht, wird der Zug in einer globalen (d.h. außerhalb der Funktion verfügbaren) Variable gespeichert.

```
function max (spieler, tiefe, position){ // Diese Funktion maximiert die
Werte für eine Position und eine Tiefe.
    chess = new Chess(position); // Erstelle ein neues Schach-Objekt für die
gegebene Position.
    if (tiefe == 0 || chess.game_over()) // Wenn das Spiel vorbei oder die
Tiefe 0 ist, gibt die Evaluierung des Brettes zurück.
        return evalPosition(position);
    let maxWert = -Infinity; // Der beste Wert. Da jeder Zug besser als
keiner ist, beginnt dies bei -unendlich.
    let züge = chess.moves(); // Speichere die legalen Züge.
    for (let i = 0; i < züge.length; i++) { // Für jeden möglichen Zug wird
das hier ausgeführt:
        chess.move(züge[i]); // Der Zug wird gezogen.
        let wert = min( //  Mit der Minimierungs-Funktion wird der Wert
festgelegt.
            -spieler, // Als Argumente werden der andere Spieler, die um 1
verringerte Tiefe und die aktuelle Position abgegeben.
            tiefe - 1,
            chess.fen(),
        );
        chess.undo(); // Der Zug wird zurückgenommen, damit ein neuer Zug
ausgeführt werden kann.
        if (wert > maxWert) { // Wenn der Wert besser als der vorherige
beste Wert ist, wird er gespeichert.
            maxWert = wert;
            if (tiefe == gewuenschteTiefe)
                gespeicherterZug = züge[i]; // Wenn wir auf der höchsten
Tiefe sind, wird der Zug auch gespeichert.
        }
    }
    return maxWert; // Der maximale Wert wird am Ende für weitere Verwendung
zurückgegeben.
}
```

Die Min-Funktion funktioniert exakt so wie die Max-Funktion, nur, dass sie minimiert, statt zu maximieren.

```
function min (spieler, tiefe, position){ // Diese Funktion minimiert die
Werte für eine Position und eine Tiefe.
    chess = new Chess(position); // Erstelle ein neues Schach-Objekt für die
gegebene Position.
```

```
    if (tiefe == 0 || chess.game_over()) // Wenn das Spiel vorbei oder die
Tiefe 0 ist, gibt die Evaluierung des Brettes zurück.
        return evalPosition(position);
    let minWert = Infinity; // Der kleinste Wert. Da jeder Zug beim
Minimieren „schlechter“ als keiner ist, beginnt dies bei unendlich.
    let züge = chess.moves(); // Speichere die legalen Züge.
    for (let i = 0; i < züge.length; i++) { // Für jeden möglichen Zug wird
das hier ausgeführt:
        chess.move(züge[i]); // Der Zug wird gezogen.
        let wert = max( //  Mit der Maximierungs-Funktion wird der Wert
festgelegt.
            -spieler, // Als Argumente werden der andere Spieler, die um 1
verringerte Tiefe und die aktuelle Position abgegeben.
            tiefe - 1,
            chess.fen(),
        );
        chess.undo(); // Der Zug wird zurückgenommen, damit ein neuer Zug
ausgeführt werden kann.
        if (wert < minWert) { // Wenn der Wert kleiner als der vorherige
beste Wert ist, wird er gespeichert.
            minWert = wert;
            if (tiefe == gewuenschteTiefe)
                gespeicherterZug = züge[i]; // Wenn wir auf der höchsten
Tiefe sind, wird der Zug auch gespeichert.
        }
    }
    return minWert; // Der maximale Wert wird am Ende für weitere Verwendung
zurückgegeben.
}
```

Es wird immer zwischen den beiden Algorithmen abgewechselt, bis die Tiefe 0 erreicht. Dann wird die nächste Möglichkeit im Spielbaum berechnet, bis der Spielbaum bis zur gewünschten Tiefe komplett analysiert wurde. Am Ende wird der von diesem Algorithmus optimal berechnete Zug gespeichert, sodass dieser vom Algorithmus gezogen werden kann.

4.2 Grafische Oberfläche und Design

Damit der JS-Code im Browser richtig ausgeführt werden kann, braucht es auch noch eine HTML-Datei (meist wird index.html verwendet, da dies die automatische Weiterleitung von den meisten Webbrowsern ist). Eine CSS-Datei ist optional, aber sorgt dafür, dass es viel besser aussieht und eine graphisch ansprechende nutzerfreundliche UI hat.

Ich werde beim HTML-Code nur das, was im <body></body>-Tag steht, eintragen, da der <head></head> fast nur Verlinkungen und Meta-Referenzen enthält.

Das hier ist der HTML-Code:

```
<div id="container">
    <label for="slider_depth">Tiefe auswählen. Warnung: Hohe
Tiefe führt zu extrem hoher Rechendauer!</label><br>
    <input type="range" id="slider_depth">
    <div id="board-container">
        <span id="board"></span>
```

```
        <progress id="prog_eval" max="20"
value="10"></progress>
    </div>
    <button id="btn_remis">Remis anbieten</button>
    <button id="btn_new-game">Aufgeben und neues Spiel</button>
</div>
```

Es ist ziemlich simpel aufgebaut, da es nur das Schachbrett und einige Knöpfe enthält. Wenn ich die gesamte CSS-Datei zeigen würde, wäre dies zu lang, weswegen ich nur die wichtigsten Teile zeige:

```
@import
url("https://fonts.googleapis.com/css2?family=Inter:wght@100;20
0;300;400;500;600;617;700;800;900&display=swap");

* {
    font-family: Inter, -apple-system,
        BlinkMacSystemFont, 'Segoe UI', Roboto,
        Oxygen, Ubuntu, Cantarell, 'Open Sans',
        'Helvetica Neue', sans-serif;
}
```

Das hier importiert die wichtigste Schriftart, Inter, und verwendet sie als Standard überall im Dokument.

```
#board {

    position: absolute;
    width: 400px;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
}
```

Hiermit wird das Brett immer in der absoluten Seitenmitte platziert. Um es für mobile Endgeräte zu verbessern, befindet sich in der JS-Datei ein Code, der das Brett auf die maximale Seitenbreite ändert, sollte das Gerät weniger als 400 Seitenpixel (1/96 eines Inchs) breit sein.

```
#prog_eval {
    display: block;
    -webkit-appearance: none;
    appearance: none;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: rotate(90deg) translate(-50%, -50%);

    width: 200px;
}

progress[value]::-webkit-progress-bar {
    background-color: #eee;
    border-radius: 2px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.25)
        inset;
}

progress[value]::-webkit-progress-value {
    background-image: -webkit-linear-gradient(
            -45deg,
```

```
                transparent 33%,
                rgba(0, 0, 0, .1) 33%,
                rgba(0, 0, 0, .1) 66%,
                transparent 66%
            ),
            -webkit-linear-gradient(top, rgba(255, 255, 255, .25), rgba(0, 0, 0, .25)),
            -webkit-linear-gradient(left, #000, #000);

    border-radius: 4px;
    background-size: 35px 20px, 100% 100%,
        100% 100%;
}
```

Und das hier gestaltet die Leiste, die anzeigt, welche Farbe in der Partie aktuell besser steht.

5. Fazit

Schach ist derzeit und wird wahrscheinlich auch in Zukunft nicht komplett von Algorithmen komplett lösbar (sein), im Gegensatz etwa zu der einfacheren Schachvariante Räuberschach[15]. Selbst, wenn man ein einzelnes Atom zum Speichern einer gesamten Schachpartie nutzen könnte, gäbe es im gesamten Universum nicht genug Atome, um alle möglichen zu speichern.

Aber Schachalgorithmen wie Stockfish, AlphaZero oder Komodo kommen sehr nah an die Perfektion des Spiels der Könige. Diese Algorithmen werden immer stärker und nutzen schon heute Daten von Millionen von Schachpartien. Menschliche Spieler können damit schon lange nicht mehr mithalten.

Dennoch merkt man, dass der Entwicklung von Schachcomputern ein Limit gesetzt ist. Die Elo-Punkte der Stockfish-Engine steigen mit jeder neuen Version etwas langsamer.[6] Es ist zwar klar, dass wahrscheinlich niemals ein Mensch auf diesem Level spielen wird, aber dennoch kommen wir mit Schachalgorithmen an die angesprochenen Grenzen.

Quellen / Literatur

[1]: https://lichess.org – „Kostenloses Online-Schach“ - Thibault Duplessis – geprüft: 17.10.2021
[2]: https://10000mates.github.io – Leo Blume (ich) – geprüft: 17.10.2021
[3]: https://stockfishchess.org – „Stockfish - Open Source Chess Engine“ – The Stockfish Foundation – geprüft: 17.10.2021
[4]: https://deepmind.com/research/open-source/alphazero-resources - „AlphaZero Resources | DeepMind“ – geprüft: 17.10.2021
[5]: https://komodochess.com/ - „Komodo Chess Engine – World Champion Chess Computer“ – ChessBase – geprüft: 17.10.2021
[6]: https://de.chessbase.com/post/monte-carlo-statt-alpha-beta - „Monte Carlo statt Alpha-Beta? | ChessBase“ – geprüft: 14.10.2021
[7]: https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1 - „Game Theory: The Minimax Algorithm Explained“ – Marissa Eppes – geprüft: 13.10.2021
[8]: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html – „Algorithms“ – Universität Stanford – geprüft: 17.10.2021
[9]: https://python.org – „Welcome to Python.org“ – Guido van Rossum + The Python Foundation – geprüft: 17.10.2021
[10]: https://pypi.org/project/chess/ - „chess · PyPI“ – Niklas Fiekas – geprüft: 13.10.2021
[11]: http://mathematics.chessdom.com/shannon-number – „The power of Shannon | ChessDom“ - ChessDom – geprüft: 14.10.2021
[12]: https://physics.stackexchange.com/questions/47941/dumbed-down-explanation-how-scientists-know-the-number-of-atoms-in-the-universe – „Dumbed-down explanation how scientists know the number of atoms in the universe?“ – StackExchange Cosmology / User Pacerier – geprüft: 17.10.2021
[13]: https://www.bbc.co.uk/news/technology-16543497 – „IBM researchers make 12-atom magnetic memory bit – BBC News“ – BBC Technology – geprüft: 16.10.2021
[14]: https://i.stack.imgur.com/8KTon.png – „12 Atoms needed to store 1 bit of data“ - IBM – geprüft: 14.10.2021
[15]: https://lichess.org/forum/general-chess-discussion/antichess-has-been-solved – „Antichess has been solved!“ – Lichess / User FischyVishy – geprüft: 17.10.2021
[16]: „Game Changer – AlphaZero’s Groundbreaking Chess Strategies and the Promise of AI“ – Matthew Sadler & Natasha Regan – geprüft: 17.10.2021