Minimax-Algorithmus (Alpha-Beta-Pruning) für Schach

Einleitung

Dieses Projekt hat das Ziel, die theoretischen Grundlagen des Minimax-Algorithmus sowie das Alpha-Beta-Pruning auf das Spiel Schach anzuwenden und dabei erste, praktische Implementierungserfahrungen mit Python zu sammeln. Der Fokus liegt dabei auf einem ausgewogenen Verhältnis zwischen theoretischer Fundierung und einer effizienten Implementierung, die trotz einfacher Konzepte funktionale und nachvollziehbare Ergebnisse liefert.

Eine zentrale Motivation ist die Vertiefung des im Studium erlernten Wissens durch die Anwendung klassischer Algorithmen der Künstlichen Intelligenz im Spielkontext. Schach bietet durch seine strategische Tiefe und klare Regeln ein ideales Testfeld für solche Ansätze und erlaubt zugleich die sukzessive Verbesserung durch zusätzliche Optimierungen.

Besonderen Wert lege ich dabei auf die Nachvollziehbarkeit und Transparenz meiner Entscheidungen bei der Implementierung und Entwicklung. Das bedeutet, dass jede Designentscheidung, von der Wahl der Evaluationsfunktion bis zur Algorithmus-Optimierung, klar begründet und dokumentiert wird. Ziel ist es, ein Projekt zu schaffen, das sowohl technisch fundiert als auch für Außenstehende verständlich und zugänglich ist.

Im Laufe des Projekts wird der Algorithmus durch verschiedene Erweiterungen wie verbessertes Move Ordering, Heuristiken zur Positionsevaluierung und spezifische Spielstrategien verfeinert. Dabei werden stets Komplexität und Ressourcenverbrauch berücksichtigt, um eine praktische und sinnvolle Balance zu finden.

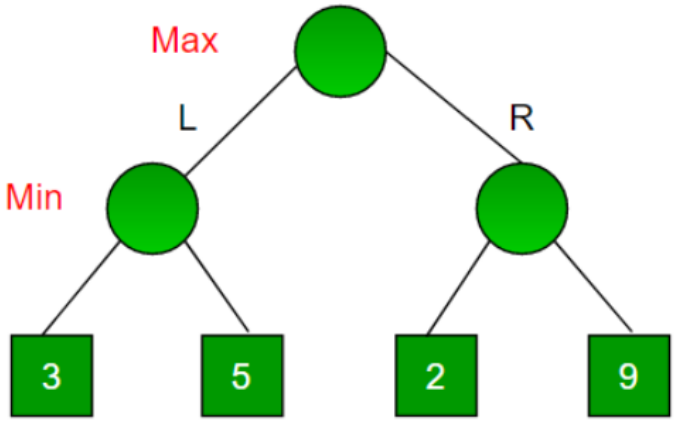
1.1 MiniMax

Ein **Minimax** Algorithmus ist ein Backtracking Algorithmus, welcher genutzt wird um den optimalen Zug für einen Spieler in einem beliebigen Spiel zu finden.
Dieser Algorithmus findet oft Anwendung in **zwei-Spieler-Spielen** wie etwa Schach oder Tic-Tac-Toe, da hier das Spiel zug-basiert stattfindet.

Bei **MiniMax** wird jedem Zustand des Spiels ein Wert zugeordnet, ist dieser Wert positiv, so scheint aktuell Spieler 1, auch **Maximizer** genannt, die überhand zu haben.
Ist der Wert des aktuellen Spielzustands negativ, so scheint Spieler 2, welcher auch **Minimizer** genannt wird, im Vorteil zu sein.

1.1.2 Beispiel

Nehmen wir an wir haben den folgenden Binärbaum gegeben:



Der **Max-Spieler** möchte also den **größten Wert** erreichen, während der **Min-Spieler** den **kleinsten Wert** erreichen möchte.
Spieler **Max** ist zuerst dran. Intuitiv würde man sagen, der Spieler sollte nach Rechts gehen, da dort der größte Wert mit 9 liegt. So würde jedoch der Spieler **Min** im nächsten Zug die 2 wählen können und würde somit den global niedrigsten Wert erreichen, das wollen wir natürlich verhindern.
So wäre in diesem Fall der beste Zug für **Max** nach links zu gehen, nicht um selbst den höchsten Wert zu erreichen, sondern um das Erreichen des niedrigsten Wertes für **Min** zu verhindern, da wir immer davon ausgehen müsse, dass der andere Spieler optimal spielt.

Aber wie bestimmen wird den Wert z.B. eines aktuellen Schachbretts?

1.2 Position evaluieren

Wir möchten nun also feststellen, welcher Spieler aktuell im Vorteil ist, bzw. wessen Gewinnwahrscheinlichkeit aktuell höher ist.
Dafür brauchen wir eine Heuristik, welche dem aktuellen Schachbrett einen Wert zuweist, basierend auf den bereits erläuterten Konzepten.
Ein einfacher Ansatz dafür wäre zum Beispiel, die Figuren selbst mit verschiedenen Werten zu belegen:

```
1       QUEEN = 900
2       ROOK= 400
3       BISHOP = 300
4       KNIGHT = 300
5       PAWN = 100
6       KING = 0
```

Alleine durch diese Werte, würde ein Algorithmus also bereits verstehen, dass es Sinn macht Figuren zu schlagen um die Punktdifferenz zu maximieren

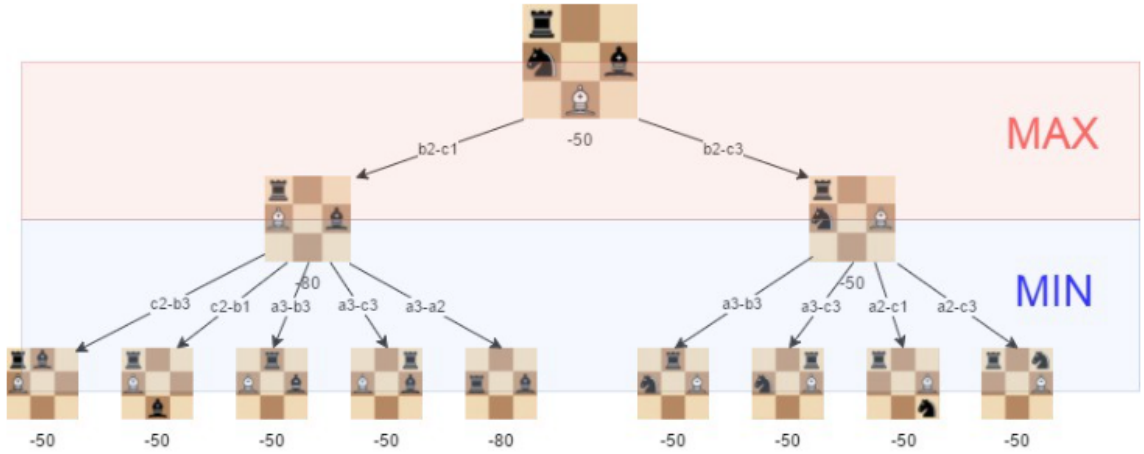
Der König selbst trägt dabei einen Wert von `0`, was zunächst nicht sehr intuitiv erscheint, der König ist doch die wertvollste Figur oder nicht?
Obwohl die Könige tatsächlich die wichtigsten Figuren eines Schachspiels darstellen, tragen sie keinen numerischen Wert. Der Wert des Königs ist in diesem Sinne unendlich, da durch den Verlust des Königs das Spiel sofort endet, er kann also auch nicht geopfert werden, was einen materiellen

Wert irrelevant macht.
Um zu verhindern, dass der Algorithmus den Wert des Königs selbst maximiert, werden stattdessen Aspekte wie etwa die Sicherheit des Königs oder eine Mattposition bewertet.
Diese werden später noch weiter behandelt

1.3 Search Tree Using Minimax

Da wir beim Schach spielen nicht nur Figuren mit dem höchsten bzw. niedrigsten Wert schlagen, sondern auch nächste Züge in Betracht ziehen müssen, eignet es sich an dieser Stelle perfekt einen Suchbaum einzuführen, welcher das Suchen nach dem besten Zug ermöglicht:

1.3.1 Beispiel



In dieser Grafik sehen wir also, dass der beste Zug `b2-c3` ist, da somit garantiert werden kann, dass im nächsten Zug minimal ein Wert von -50 erreicht werden kann, während mit `b2-c1` Schwarz im nächsten Zug den Wert des Brettes auf -80 verringern könnte.
Wichtig ist zu betrachten, dass hier die Qualität des Algorithmus maßgeblich von der Tiefe des Suchbaums abhängt.
$\rightarrow$ Je mehr Züge im Voraus betrachtet werden, desto besser der Algorithmus.
Das ist jedoch extrem Ressourcen-aufwendig.

1.4 Alpha-Beta Pruning

Bei dem **Alpha-Beta Pruning** wird durch das strategische Ausschließen von bestimmten Teilbäumen eine deutlich bessere Laufzeit erreicht, da nicht im jedem Fall jeder Zug "fertiggedacht" werden muss, falls klar ist, dass dieser auf jeden Fall schlechter als ein Zug ist, den wir bereits betrachtet haben.

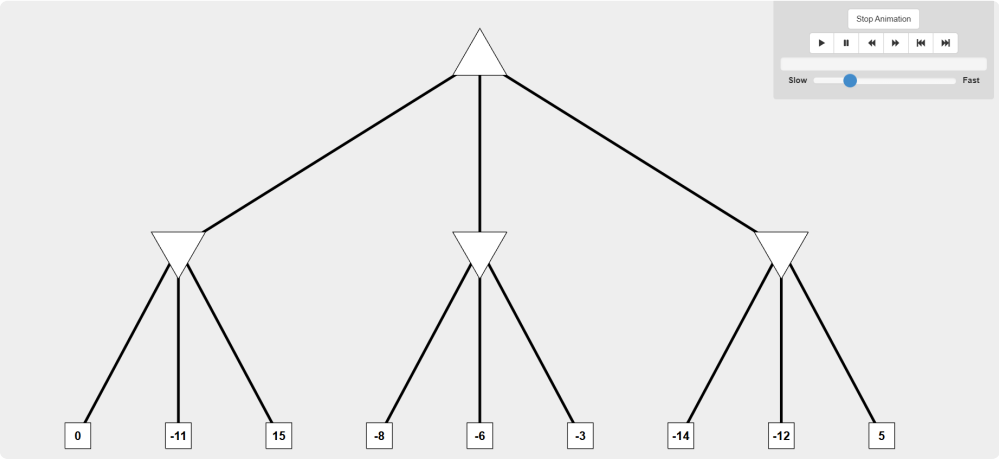
1.4.1 Beispiel

Wir definieren hier zunächst zwei Werte:

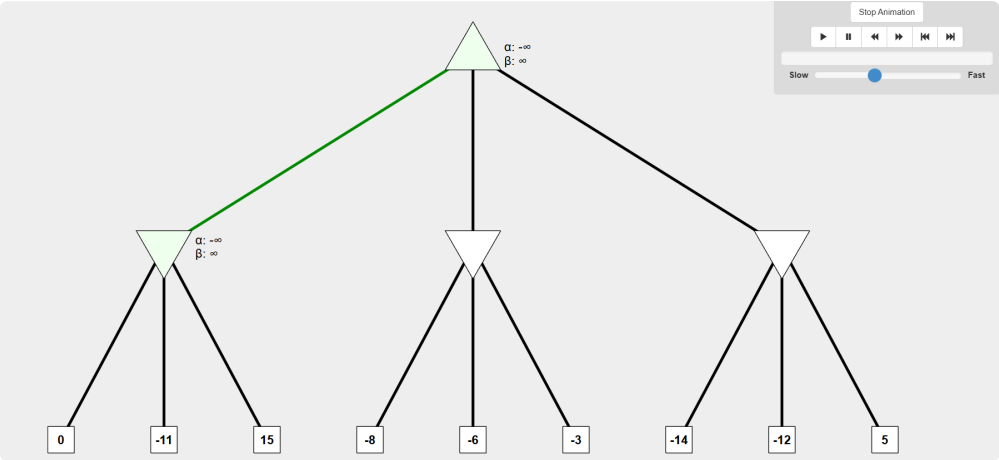
- **Alpha**(α): Die Mindestpunktzahl die der Maximizer garantiert erreichen kann.
- **Beta(**β**)**: Die Höchstpunkzahl die der Minimizer garantiert erreichen kann.

Gegeben sei der folgende Suchbaum, wobei die Werte der Blätter eine relative Wertung der möglichen Züge darstellen.

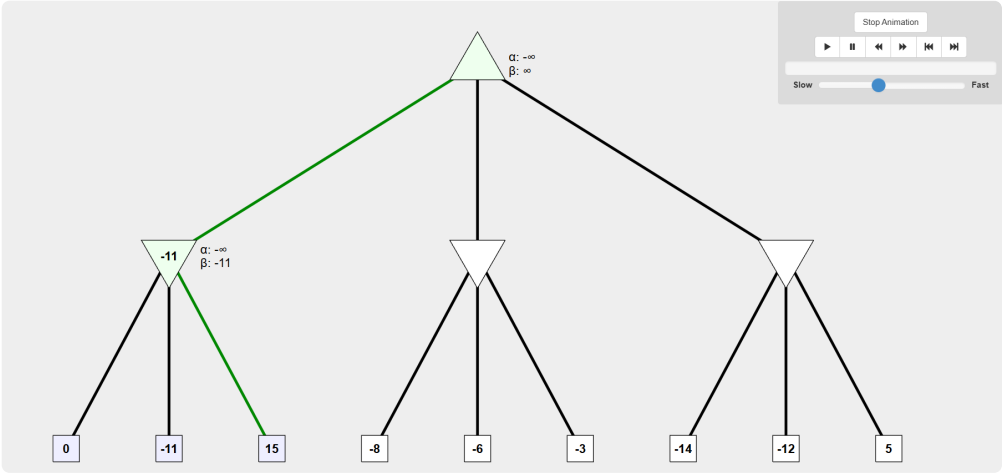
Der Maximizer ist am Zug.



Als nächstes traversieren wir den Baum in **Pre-Order** und setzten bei jeden nicht-Blattknoten die α und β-Werte zunächst auf $-\infty$ bzw. $+\infty$.



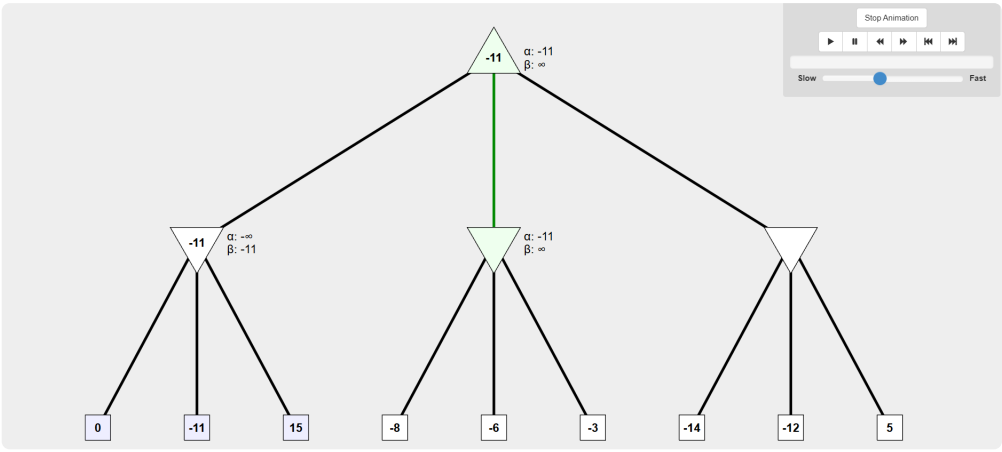
Da der Maximizer am Zug ist, sind die Blattknoten die Wertungen der möglichen **Antwortzüge des Minimizers**.
Der Minimizer sucht in den Kindknoten nach dem kleinsten Wert, da dieser der für ihn beste Zug darstellt, in diesem Fall also -11.
Der Elternknoten bekommt also einen β-Wert von -11.



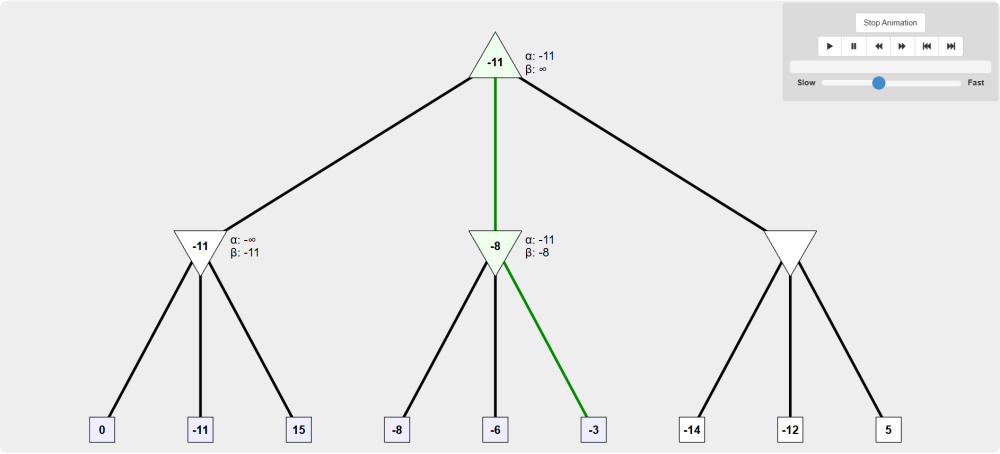
Nun aktualisiert sich der α-Wert der Wurzel.

Warum?

Bei diesem Vorgehen muss immer davon ausgegangen werden, dass beide Seiten optimal spielen, also den jeweils besten Zug wählen.
Wenn wir der Annahme folgen, so wird der Minimizer auf jeden Fall mit dem Zug antworten, welcher zu einer Bewertung von -11 führt, wenn der Maximizer den Zug A gespielt hat.
Zu diesem Zeitpunkt ist also die Mindestanzahl an relativen Punkten die der Maximizer erreichen kann -11 indem er Zug A spielt und davon ausgeht, dass der Minimizer den Zug mit Wertung -11 spielt.
$\rightarrow \alpha = -11$

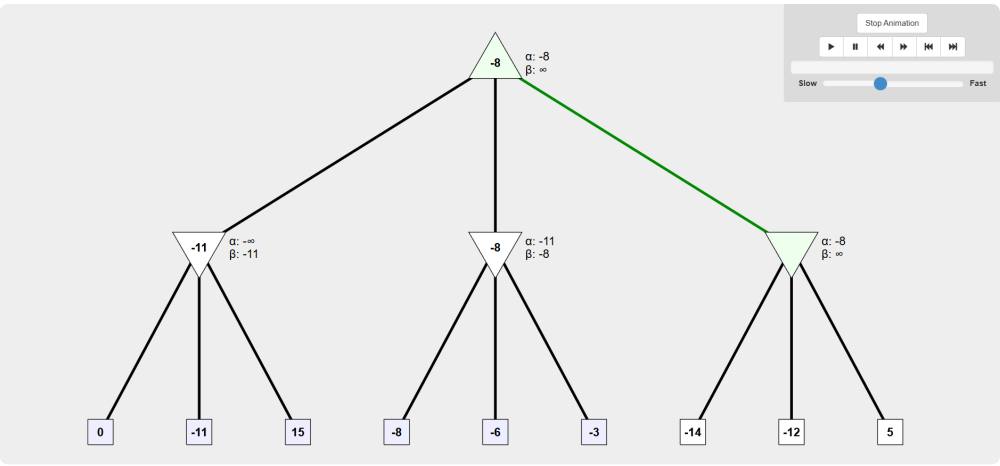


Dieses Vorgehen wiederholen wir nun Iterativ für alle Knoten des Suchbaums, die nächsten Schritte passieren hier also analog.



Betrachten wir nun die erneute Aktualisierung des α-Werts der Wurzel.
Wir erkennen, dass der Minimizer nach dem Spielen von Zug B, den Wert maximal auf -8 verringern kann.
Der Maximizer weiß nun also, dass die beste Antwort von dem Minimizer auf Zug B, eine für den Minimizer schlechtere Position bietet, als die beste Antwort auf Zug A.

Nun ist also die Mindestanzahl der relativen Punkte die der Maximizer erreichen kann auf -8 gestiegen.
Bei einer Tiefe von zwei ist der beste Zug nun also Zug B.

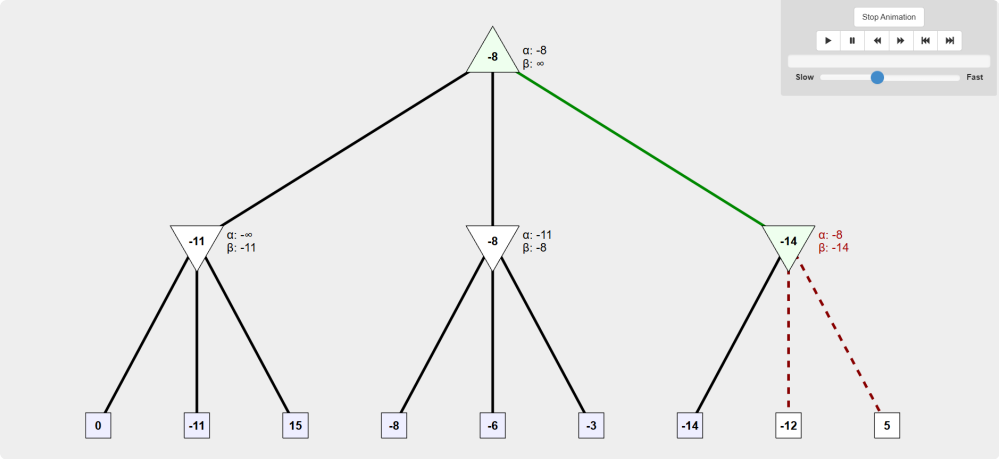


Wie wir sehen wurde bis jetzt keine Verbesserung zu unserem ursprünglichen, einfachen MiniMax Algorithmus erreicht. Es musste jeder Knoten betrachtet werden und mit dem α bzw. β-Wert des Elternknoten verglichen werden.

Schauen wir nun aber auf die letzte Iteration, also die Antwortzüge auf Zug C, so erkennen wir bereits bei dem betrachten des ersten Blattknotens, dass der Minimizer auf den Zug C eine Antwort kennt, welche zu einem relativen Wert von -14 führt.
Bereits jetzt weiß der Minimizer, dass er diesen Zug nicht spielen wird, da $-8 = \alpha \geq \beta = -14$.

Einfach gesagt:
Wenn der Maximizer den Zug C spielt, erreicht der Minimizer weniger Punkte, als wenn er den bis jetzt besten Zug, Zug B spielt. Genau das wird durch $\alpha \geq \beta$ impliziert.
Daher schließt der Maximizer diesen Zug schon aus, bevor wir überhaupt die restlichen Antwortzüge betrachtet haben.

$\Rightarrow$ Wenn $\alpha \geq \beta$ für einen Knoten x, so werden alle Teilbäume x_i des Knotens x, bei dem diese Bedingung auftritt, ignoriert bzw. **abgeschnitten**. Hier kommt es also zum **Pruning** (Cut-Off), welches dem Algorithmus seinen Namen verleiht.



---

2 Weitere Verbesserungen

Nachdem wir uns nun mit den Grundlegenden Konzepten vertraut gemacht haben, können wir Anfangen die Suche nach möglichen Zügen sowie die Evaluierungs-Funktion zu verbessern um somit die Spielqualität und Laufzeit unseres Algorithmus zu erhöhen.

Sowohl in der Suche nach Zügen als auch in der Evaluierung dieser, gibt es eine beinahe unbegrenzte Möglichkeiten der Optimierung und Verfeinerung des Algorithmus.
Da dieses Projekt aber, wie bereits in der Einleitung beschrieben, sich vor allem mit den Grundlagen von Schachalgorithmen beschäftigt soll und zusätzlich das bereits im Studium erlernte Wissen vertiefen soll, werden wird uns hier auf ein paar Grundlegende Optimierungen festlegen.
Die Wahl dieser beruht dabei immer auf einem ausgewogenen Verhältnis zwischen Komplexität und Optimierung, sodass bereits durch diese wenigen Konzepte eine deutliche Verbesserung der eben genannten Aspekte möglich ist. Sie bieten somit eine solide Grundlage für spätere Erweiterungen oder Optimierungen.

2.1 Improved Search

Im Folgenden wollen wir die Suche in unserem Suchbaum weiter optimieren.
Wir werden uns hierbei auf zwei Konzepte konzentrieren, das **Move Ordering** und das **Iterative Deepening**, um sowohl die Laufzeit weiter zu verbessern, als auch die Suche an sich zu optimieren.

2.1.1 Move Ordering

Durch das **Move Ordering** wollen wir die Effizienz das **Alpha-Beta Pruning** verbessern, indem wir die Züge auf jeder neu betrachteten Ebene des Suchbaums zunächst durch eine weniger komplexe Bewertungsfunktion vor sortieren und somit die Zugqualität bereits im vorhinein abschätzen.

Move Ordering ist mit Iterative Deepening besonders effektiv, dieses Konzept werden wir im nächsten Abschnitt behandeln.

Somit können bereits evaluierte Stellungen später nicht nur nach einer einfacheren Bewertungsfunktion sortiert werden, sondern direkt auf Basis ihres im vorherigen Durchlauf berechneten Wertes, was die Qualität des Move Orderings in späteren Iterationen erhöht.

2.1.2 Iterative Deepening

Bei dem **Iterative Deepening** führen wir unseren bekannten MiniMax-Algorithmus mit Alpha-Beta Pruning iterativ mit immer größerer Suchtiefe aus.
Durch das eben beschriebene **Move Ordering** erhalten wir in jeder Iteration eine bessere Abschätzung der Qualität der Züge, da die Informationen aus vorherigen Iterationen genutzt werden um die Züge bei neuen Iteration zu sortieren.
Somit können wir mit den selben Ressourcen eine tiefere Suche durchführen, da die Ressourcen vor allem zur tiefen Evaluierung von besseren Zügen genutzt wird und durch die Sortierung die meisten Züge effizient ausgeschlossen werden können (**Pruning**).

Bei einer Schachstellung mit einer Suchtiefe von vier Halbzügen (jeder Spieler zieht zweimal) und einem durchschnittlichen Verzweigungsfaktor von 36 ergeben sich bei einer vollständigen Minimax-Suche über 28 Millionen zu bewertende Endknoten. Durch den Einsatz von Alpha-Beta-Pruning reduziert sich diese Anzahl auf etwa 2 Millionen Knoten. Wird zusätzlich eine Zugsortierung implementiert, sinkt die Anzahl der bewerteten Knoten sogar auf ungefähr 128.000. Dies entspricht einer Reduktion von über 99 %.

Ein weiterer Vorteil des Iterative Deepening ist die Möglichkeit, eine Abbruchbedingung zu definieren, anstatt eine feste Tiefe anzugeben. Dadurch kann nach jeder Iteration das aktuell optimale Ergebnis zurückgegeben werden, falls z.B. eine Zeitschranke überschritten wird.

Transposition Tables

Transposition Tables werden genutzt um bereits evaluierte Schachpositionen zu speichern, somit kann erheblich Rechenzeit und Ressourcen gespart werden.
Genauer wird dabei jede Schachposition in einer Hash-Tabelle gespeichert und vor jeder Evaluierung geprüft, ob diese Position schonmal betrachtet wurde. Grade mit Iterative Deepening ist diese Verbesserung ausschlaggebend, da hier tatsächlich "vorprogrammiert" ist, dass gleiche Positionen mehrfach evaluiert werden.

2.2 Improved Evaluation Function

Durch das Anpassen und Verbessern unserer Evaluations Funktion verbessern wir die Zugqualität unseres Algorithmus.

2.2.1 Piece Square Tables

Die **Piece Square Tables** erhöhen bzw. verringern den Wert einer Figur anhand ihres Standpunktes auf dem Schachbrett.

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

Beispiel

Hier sehen wir zum Beispiel, dass eine Dame im Zentrum des Schachbrettes mehr Wert ist als am Rand oder vor allem in den Ecken, da hier die Möglichkeit an Zügen im Allgemeinen deutlich reduziert wird.
Die Piece Square Tables können je nach Spielabschnitt angepasst werden, somit sieht der Piece Square Table einer Dame im Anfangsspiel und im Endspiel unterschiedlich aus und somit verändert sich auch das Verhalten der spezifischen Figuren über den Verlauf des Spiels.

2.2.2 Center Control

Das **Center Control** erweitert die Piece Square Tables.
Während bei den Piece Square Tables für jede Figur spezifisch angegeben wird, wo diese besonders Wertvoll ist, sollen hier allgemeinere Regeln aufgestellt werden um den Algorithmus bei seinen Zügen auf die Mitte des Feldes (`d4` , `d5` , `e4` , `e5`) zu konzentrieren.

Dynamic Control

Mithilfe der **Dynamic Control** können wir ein solches Verhalten erreichen.
Für jeden möglichen Zug einer Figur auf den zentralen Feldern gibt es eine positive Bewertung, das bedeutet, grade Figuren wie Dame, Springer und Läufer sind im Zentrum besonders wertvoll. Somit werden Züge, welche bewegliche Figuren in das Zentrum verlagern häufiger gespielt.

2.2.3 King Safety

Bei dem Prinzip der **King Safety** geht es darum, dem Algorithmus beizubringen, dass der König sicher bleiben soll und vor allem wie das erreicht wird.
Hierzu gibt es zahlreiche Ansätze. Wir werden uns genauer mit **Pawn Shield**, **King Tropism** und **Virtual Mobility** beschäftigen, da diese sich vor allem als Grundlegende Regeln besonders gut geeignet sind und später beliebig erweitert werden können.

Pawn Shield

Grade nach einer Rochade ist es wichtig, dass der König von den eigenen Bauern geschützt bleibt. Durch die negative Bewertung offener Linien und die positive Bewertung der Bauern direkt vor einem König, lernt der Algorithmus, dass diese Bauern wichtig sind und nur äußerst selten bewegt werden sollten.

Virtual Mobility

Das schon beschriebene Konzept des **Pawn Shield** kann gut von der **Virtual Mobility** ergänzt werden.
Hierbei wird der König kurzeitig in eine Dame verwandelt, da diese die beweglichste Schachfigur darstellt. Durch die Anzahl der möglichen Züge dieser Dame kann eine grobe Aussage über die Erreichbarkeit des Königs getroffen werden.
Die Idee ist also, je mehr Züge die temporäre Dame zur Verfügung hat, desto einfacher ein Angriff auf den eigenen König.
Werten wir die möglichen Züge der Dame also negativ, so wird ein willkürliches öffnen von Linien auf den König verhindert.

King Tropism

Bei dem **King Tropism** wird die Bewertung der eigenen und gegnerischen Figuren auf Basis der ausgehenden Gefahr sowie der Distanz zum eigenen König angepasst.
Haben wir zum Beispiel eine Dame in direkter Nähe zum König, so wird dieser eine höhere Gefahr zugewiesen wie etwa einem Springer, beide erhalten dennoch durch ihre Nähe zum König einen relativen Wert zugewiesen.
Bei eigenen Figuren verhält sich das ähnlich, eine Dame sichert in der Nähe des Königs mehr Linien und Felder ab als ein Springer, somit erhöht sich hier der Wert.

In diesem Projekt werden wir jedoch **King Tropism** nur im Bezug auf den Druck auf den gegnerischen König verwenden, da **Virtual Mobility** und **Pawn Shiled** bereits für ein, diesem Projekt entsprechendes, defensives Verhalten sorgt.

2.2.4 Evaluation of Pieces

Durch die **Evaluation of Pieces** können wir jede Figur selbst nochmal auf bestimmte Kriterien überprüfen und somit Figur-spezifische Regeln festlegen.

Die Bewertung der Stellungen der Figuren soll in unserem Projekt vor allem durch die **Piece Sqaure Tables** vorgenommen werden.
Dennoch möchte ich ein paar weitere, einfache Regel implementieren, welche das Stellungsspiel unseres Algorithmus um einiges verbessern wird.

Pawn

Wir wollen den Algorithmus dazu anreizen, sogenannte **Centerswaps** durchzuführen.
Hierbei erhalten Doppelbauern, welche durch das Schlagen einer Figur auf der $d-$ oder $e-$Linie entstehen, einen Bonus.
In Kombination mit den **Piece Square Tables** werden somit Doppelbauern im Zentrum besonders belohnt.

Knight

Oft werden Knights höher gewertet wenn sie einen sogenannten **Outpost** bilden.
Hier sind die Knights zentral im eigenen oder auch im gegnerischen Teil des Feldes platziert und

werden von einem Pawn beschützt.
Wir wollen an dieser Stelle diese Regeln etwas allgemeiner Formulieren und Werten im generellen Knights höher, welche von einem Pawn verteidigt werden.

Bishop

Für den Bishop werden wir versuchen einen so genannten **Bad Bishop** zu verhindern.
Ein Bad Bishop ist in seiner Mobilität Richtung Gegner durch die eigenen Bauern beschränkt, dies ist also vor allem im Eröffungsspiel und in der Mitte des Spiels relevant.
Wir wollen also durch eine clevere Anpassung des Wertes eines Bishops, falls dieser von seinen eigenen Bauern zugestellt ist, dies Verhindern und somit auch die Entwicklung der Bishops im Eröffnungsspiel fördern.

Rook

Rooks stellen die zweit stärksten Figuren im Spiel dar und sind grade im späteren Verlauf des Spiels sehr wichtig.
Um eine ungefähre Einschätzung des Spielstands zu bekommen, erhöhen wir den Wert der Rooks in Abhängigkeit der Bauernanzahl auf dem Spielfeld, somit simulieren wir die steigende Bedeutung der Rooks im späteren Verlauf des Spiels.

Queen

Für die Queen werden wir eine Strafe für zu frühes Entwickeln einführen, somit spielt unser Algorithmus im Bezug auf die Queen zunächst etwas defensiver.
Da wir bereits bei den Rooks den Spielfortschritt anhand der Bauernanzahl definieren, werden wir hier die selbe Metrik anwenden.
Um dieses Vorgehen weiter zu ergänzen wollen wir zudem den relativen Wert der Queen im Bezug auf den **King Tropism** erhöhen, somit Entwickelt sich die Dame nicht nur später sondern ist auch auf die Verteidigung des Königs fokussiert.

King

Bei dem King verzichten wir auf zusätzliche Regeln, da das Prinzip der King Safety sowie die Piece Square Tables das für den Umfang dieses Projektes gewünschte Verhalten bereits implementieren.

Quellen und weiterführende Links

1. Minimax-Algorithmus Einführung bei GeeksforGeeks
2. Einführung in Alpha-Beta Pruning
3. Algorithms Explained – minimax and alpha-beta pruning
4. Schach-KI Schritt für Schritt (FreeCodeCamp)
5. Chess Programming Wiki
6. Alpha-Beta Pruning Practice
7. Alpha-Beta Pruning Wikipedia