# Das Schachprogramm Albatros

Daniel Grévent

7. Juli 2022

# Inhaltsverzeichnis

# 1 Einleitung

Schach wurde in dem letzten Jahrhundert als Testplattform für künstliche Intelligenz verwendet, da die Spielregeln relativ einfach sind, während das Spiel selbst komplex ist. Die Anfänge von Schachprogrammen lassen sich auf den Anfängen des Computers zurückverfolgen. Konrad Zuse, der Erfinder des ersten Digitalen Computers, entwickelte zwischen 1943 und 1945 das weltweit erste Schachprogramm. Die ersten Grundsätze der Theorie hinter Schachprogramme legte Claude Shannon, in 1950 mit Programming a Computer for playing Chess. Mit der Zeit wurden Computer immer leistungsstärker und Schachprogramme wurden immer effizienter. Ab 1997, als IBM Deep Blue, gegen den damaligen Schach Weltmeister Garry Kasparov antritt und ihn besiegte, wurde es klar, dass Computer besser Schach spielen konnten als Menschen. Die Frage, die sich dabei stellen lässt ist, wie funktionieren diese Programme?

Ich versuchen in diesem Dokument, anhand des selbstentwickelten Schachprogramms Albatros, zu bewerten welche Algorithmen wichtig sind um Schach auf einem übermenschlichen Niveau zu spielen. Ein weiteres Ziel ist es, dass Albatros an den Schachprogramm Weltmeisterschaften teilnimmt.

## 1.1 Gliederung

In diesem Dokument werde ich mit der generellen Theorie für zwei Spieler Spiele anfangen, und dann mich mit immer Schach spezifischeren Algorithmen befassen. Dann werde ich das Schachprogramm Albatros vorstellen. Später werde ich mich mit Tests und Ergebnisse befassen, um schlussendlich mit dem Fazit zu enden.

# 2 Allgemeine Theorie(Typ A)

## 2.1 Shannons Schachprogramm Definition

Claude Shannon hat sehr früh versucht Schachprogramme zu kategorisieren. Er erfand somit zwei verschiedene Kategorien Typ A und Typ B. Der Typ A wird als sogenannter ”Brute Force”Typ beschrieben, da er versucht alle mathematisch notwendigen Stellungen zu analysieren. Der Typ B Algorithmus hingegen ähnelt von der Funktionsweise her der Denkweise eines Menschen. Wenn ein menschlicher Großmeister Schach spielt analysiert er nicht alle möglichen Stellungen. Er zieht ein paar ihm gut erscheinende Züge in Erwägung und versucht für jeden Zug einen Gegenzug zu finden. Dazu bewerten Menschen Stellungen oft viel besser als Computer, da sie auch die Langzeit Effekte von Stellungen in Erwägung ziehen, was für klassische Computer Algorithmen unmöglich wäre. Heutzutage gibt es nur ein paar Schachprogramme die völlig nach Typ B funktionieren. Diese Programme spielen oft zwar sehr gut, wurden aber nur vor kurzem erfunden und sind sehr aufwendig, da sie sehr große neuronale Netzwerke benötigen. Die meisten sehr gute Schachprogramme würden sich als Typ AB definieren lassen. Sie basieren auf Typ A Algorithmen, verwenden aber verschiedene Typ B Algorithmen die versuchen schlechte Züge und Stellungen zu beseitigen.

## 2.2 Min Max

Der Min Max Algorithmus wird zur Ermittelung von optimalen Spielstrategien für zwei Spieler Spiele verwendet. Dabei wird das Spiel als Baum dargestellt, bei dem jede Stellung als Knotenpunkt repräsentiert wird. An einem Knotenpunkt sind entweder andere Knotenpunkte mittels eines Zuges verbunden oder, falls der Knoten keine möglichen Züge mehr hat, wird dieser wie folgt bewertet:

| | |
|---|---|
| 1 | der erste Spieler gewinnt |
| 0 | keiner der beiden Spieler gewinnt |
| -1 | der erste Spieler verliert |

Die zwei Spieler werden Min und Max genannt. Max ist der erste Spieler, dadurch spielt er immer den Zug mit der höchsten Wertung. Dagegen spielt Min immer nach Max, und spielt immer den besten Zug für sich, oder, in anderen Worten, den schlechtesten Zug für Max. Um den jeweiligen besten Zug zu ermitteln, wird mit den bewerteten Knoten angefangen. Jeder anschließende Knoten wird anhand der Bewertung des besten untergeordneter Knotens, je nach derzeitigem Spieler bewertet. Der Algorithmus würde in Pseudocode wie folgt aussehen:

```
1  int min_max(position, to_play, int depth){
2      int current_value = 0, best_value = 0;

4      if(position.is_terminal || depth == 0){
5          return evaluate(position);
6      }
7      else{
8          if(to_play.is_max){
9              best_value = -infinity;
10         }
11         else{
12             best_value = +infinity;
13         }

15         while(position.moves_left){

17             position.play_new_move();

19             current_score = min_max(position, to_play.other_player(), depth - 1);

21             position.undo_last_move();

23             if(to_play.is_max){
24                 if(current_score > best_value){
25                     best_value = current_score;
26                 }
27             }
28             else{
29                 if(current_score < best_value){
30                     best_value = current_score;
31                 }
32             }
33         }
34     }
35     return best_value;
36 }
```

Die Komplexität eines Min Max Baums kann mit zwei Variabeln definiert werden, die sich in jedem Spiel unterscheiden.Die erste variabel wäre der sogennante Verzweigungsfaktor, also wie viele Züge es im Durchschnitt nach einem Knoten gibt (in Schach sind es ungefähr 35). Die zweite wäre die Tiefe, oder wie viel Züge im Durchschnitt zu einer Endstellung führen (in Schach ungefähr 40 Züge). Somit wäre die Komplexität des Baums ungefähr $35^{80} = 10^{120}$. Die Anzahl wird auch Shannons Nummer genannt. Das

sind viel mehr potenzielle Spiele als es Atome im Universum gibt, daher gilt Schach als nicht lösbar oder in anderen Worten wir wissen nicht wie das perfekte Schachspiel aussieht. Da man nicht alle möglichen Stellungen berechnen kann, definiert man eine maximale Anzahl an Zügen, die von dem Programm betrachtet(eigentlich halbe Züge = depth Parameter) werden. Falls man mit einer Stellung endet, die nicht final ist, wird diese approximativ mit einer heuristischen Funktion bewertet(im Code evaluate()).

## 2.3 PSqT Bewertungsfunktion

Eine einfache heuristische Funktion zählt die Werte der Figuren des einen Spielers zusammen und subtrahiert diesen Wert von dem des anderen Spielers. Die verschiedenen Spielfiguren werden unterschiedlich stark bewertet. Dazu werden zwei weitere Eigenschaften des Schachspiels zunutze gemacht. Erstens, die Spielfiguren sollten je nach Figurentyp eher auf bestimmte Felder platziert werden(z.B. der König sollte eher rochieren): das ist das wofür PSqT (piece square tables) steht. Zweitens, es gibt in einem Schachspiel verschiedene Phasen, die je nach Anzahl an verbleibenden Figuren auf dem Schachbrett ermittelt werden. Die erste Phase ist die Eröffnung, in der die Spieler ihre Figuren möglichst gut platzieren sollten. Die zweite Phase ist das Mittelspiel, in dem die Spieler versuchen ein Vorteil mittels eines Kampfes zu gewinnen. Die letzte Phase, das Endspiel, ist die Phase in der, der bessere Spieler versucht seine Stellung in einem Sieg zu konvertieren. Da in diesen verschieden Phasen, die Figuren anders platziert sein sollten, werden die Werte der Figuren linear interpoliert. Die Funktion in meinem Programm entspricht von den Werten her, denen der Bewertungsfunktion des Schachprogramms RofChade.

## 2.4 Alpha-beta pruning

Damit ein Schachprogramm möglichst gut spielt, gibt es nur zwei Möglichkeiten, entweder die Bewertungsfunktion wird verbessert, oder es sucht tiefer, damit das Programm taktische Zugfolgen besser erkennt. Der klassische Weg ist es möglichst tief zu suchen, da eine gute Bewertungsfunktion schwierig zu programmieren ist. Eine Methode um weniger Knoten zu analysieren, und somit tiefer zu suchen ist der Alpha Beta Algorithmus.

```
1  int alpha_beta(position, int alpha, int beta, int depth){
2      if(position.is_terminal || depth == 0){
3          return evaluation(position);
4      }

6      while (position.moves_left){
7          position.play_new_move();

9          int current_value = -alpha_beta(position, -beta, -alpha, depth-1);

11         position.undo_last_move();

13         if(current_value > alpha){
14             alpha = current_value;

16             if(value >= beta){
17                 return beta;
18             }
19         }
20     }
21     return alpha;
22 }
```

Die Idee in der Alpha Beta Suche ist das jeder Spieler in dem Min Max Baum(also Min und Max) dieselbe Aktionen durchführt, aber aus einer anderen Perspektive. Dadurch wird die Funktion etwas vereinfacht. Dazu gibt es zwei Variabeln alpha und beta. Alpha ist der Wert der derzeit bestmöglichen bekannten Zugfolge von Max(der Spieler der gerade spielt). Beta ist der Wert der derzeit besten bekannten Zugfolge für Min(der Gegner). Der Grund dafür, dass weniger Knoten besichtigt werden, ist das man in manchen Fällen schon weiß, die Bewertung von den nächsten untergeordneten Knoten, die des derzeitigen Knotens nicht verbessern kann. Der Grund dafür ist, dass der andere Spieler Zugfolgen nicht akzeptieren wird, die schlechter sind als seine derzeitige beste Zugfolge. Da für den derzeitigen Spieler die beste Zugfolge in dem Knoten auf jeden Fall zu schlecht ist, kann man in diesem Knoten aufhören zu suchen.

## 2.5 Quiescence search

Bis jetzt gab es ein Problem bei der Bewertung unserer Schachstellungen. Das wird an folgendem Beispiel erkennbar: Stellen wir uns vor, dass in unserem letzten Zug vor der Bewertung, die Dame einen Bauern geschlagen hat. Dadurch würde die Stellung eher gut bewertet werden, da wir einen Bauer Vorsprung haben. Würde der Gegner mit einem weiteren Bauern die Dame schlagen, wäre die Stellung schlecht, da wir unsere Dame gegen einen Bauern verloren haben. Dieses Problem wird auch Horizont Effekt genannt, da das Programm keine weiteren Züge sieht als bei seinem definierten Horizont. Damit das Problem möglichst wenig Einfluss auf die Spielstärke des Programms nimmt, werden anstatt direkt die Stellung zu bewerten nur alle taktischen Züge (Züge in denen entweder ein König in Schach ist oder Züge in denen eine Figur genommen wird) rekursiv mit dem Alpha Beta Algorithmus gesucht. Da die Taktischen Züge nicht unbedingt die besten Züge sind werden die Stellungen auch direkt bewertet. Da davon ausgegangen wird das in jeder Stellung (außer im Zugzwang) es einen Zug gibt der die Bewertung der Stellung verbessert, ist die einfache Bewertung eine untere Grenze für die Bewertung der Stellung. Diese Idee wird null Zug Observation genannt. Wenn die beste taktische Zugfolge schlechter als die direkte Bewertung ist, wird die direkte Bewertung zurückgegeben. Die Quiescent search Suche, wird dann immer anstatt der direkten Bewertung gemacht. Die quiescence search würde als Programm ungefähr wie folgt aussehen:

```
1  int quiescence_search(position, int alpha, int beta){
2      int standing_pat = evaluation(position);
3
4      if(standing_pat > alpha){
5          alpha = standing_pat;
6
7          if(alpha >= beta){
8              return beta;
9          }
10     }
11
12     while(position.tactical_moves_left){
13
14         position.play_new_move();
15
16         int current_value = -quiescence_search(position, -beta, -alpha);
17
18         position.undo_last_move();
19
20         if(current_score > alpha){
21             alpha = current_score;
22
23             if(alpha >= beta){
24                 return beta;
25             }
26         }
27     }
```

```
28        return alpha;
29    }
```

## 2.6 Zugreihenfolge

Der alpha beta Algorithmus, sucht bei gleicher Suchtiefe weniger Stellungen als der min max Algorithmus, da er Stellungen, die die derzeitig beste Zugfolge nicht verändern können ausschließt. Damit der Algorithmus möglichst viele unwichtige Stellungen ausschließt ist es vorteilhaft, zu erst die besten Stellungen zu examinieren. Man versucht also die Reihenfolge der Züge so zu verändern, dass die Züge die wahrscheinlich am besten sind am Anfang der Suche examiniert werden. Dazu werden folgende Algorithmen verwendet:

### 2.6.1 MVV-LVA

Ein wichtiger Algorithmus ist zum Beispiel MVV-LVA(Most Valuable Victim-Least Valuable Attacker). Dem Algorithmus liegen drei Ideen zugrunde. Erstens sind taktische Züge am interessantesten. Zweitens, am interessantesten sind taktische Züge in denen eine Figur mit einem hohen wert geschlagen wird(z.B. eine Dame). Letztens, wenn mehrere Figuren eine Figur mit gleichem Wert schlagen können, sollte man eher erst den Zug mit der Figur examinieren, die den kleinsten Wert hat.

### 2.6.2 Killer heuristic

In Schach Stellungen gibt es im Durchschnitt mehr Züge die nicht taktischer Natur sind, als Züge taktischer Natur. In der Killer heuristic werden nicht taktische Züge in eine Liste die abhängig von der tiefe des derzeitigen Knotens ist gespeichert, wenn sie einen beta Cutoff(der derzeitige Wert ist so gut dass er von dem Gegner nicht angenommen wird) verursachen. Es werden gewöhnlicherweise zwei Killer Züge(nur die Züge, nicht die Stellungen) gespeichert. Wenn in einer weiteren Stellung der selben Suchtiefe, derselbe Zug möglich ist, wird dieser nach den taktischen Zügen geordnet.

### 2.6.3 Countermove heuristic

Bei der Countermove heuristic wird davon ausgegangen, dass wenn der erste Spieler einen Zug spielt, sein Gegner einen natürlichen Gegenzug hat. Diese Züge werden nach den killer Zügen geordnet.

### 2.6.4 History heuristic

Die Hisory Tabelle enthält alle möglichen Züge. In der History Tabelle wird ein Wert, die sogenannte Geschichte des Zuges, verändert immer wenn es einen beta Cutoff gibt. Dazu wird für alle zuvor gesuchten Zügen ihre Geschichte um einen bestimmten Wert verschlechtert (gewöhnlicherweise Suchtiefe * Suchtiefe) und für den Zug der den Cutoff erreicht hat wird sie um denselben wert verbessert. In Albatros gibt es drei andere History Tabellen: Countermove History (eine Mischung zwischen normaler history und counter move heuristic), followup history (die selbe Idee wie bei countermove history,

aber anstatt den letzten Zug des Gegners zu verwenden, wird der letzte Zug des derzeitigen Spielers benutzt) und capture history(in Albtros ein Teil von MVV-LVA, da dies empirisch besser war). Die History Züge werden Nach den Countermove Zügen geordnet.

## 2.7 Transpositionstabelle

Die Transpositionstabelle ist eine Art Hashtabelle. Sie wird verwendet um schnell Informationen zu früheren Suchen der selben Stellung zu finden. Die Informationen die in dem Programm gespeichert werden sind die Suchtiefe, die Bewertung der Stellung, wie die Suche bei der Stellung ausgegangen ist(um zu wissen ob der Wert aufgrund eines Cutoffs entstanden ist), der beste Zug in der Stellung wird ebenfalls gespeichert(um ihn als ersten Zug zu suchen). Sie wird am Anfang des Suchprozesses mit der Anzahl an Stellungen die sie maximal speichern kann initialisiert. Die Hash schlüssel werden mit der sogenannten Zobrist Hash Methode generiert. Der Zobrist Hash ist ein Algorithmus, der speziell für Schach Stellungen erfunden wurde. Die Idee ist folgende: jede Spielfigur hat einen Figurentyp (z.B. schwarzer Läufer) und einen Platz (z.B. e4). Es werden also allen möglichen Typ/Platz Paaren, am Start des Programms, einen zufälligen Wert zugewiesen. Die Berrechnung des Schlüssels erfolgt indem je nach Stellung für alle Typ/Platz Paare auf dem Brett der Wert ausgewiesen und zu dem Gesamtwert mit einem XOR hinzugefügt wird. Wenn in der Stellung Schwarz spielt wird zu dem Endwert ein spezifischer Wert mithilfe eines XORs zugefügt.

## 2.8 Principal variation search

Principal variation search(auch pv-search genannt) ist eine Verbesserung der alpha beta Suche. Die Grundidee von pv-search ist, dass man bei der Suche nur die beste Zugfolge sucht. Wenn sie gefunden wurde, muss man nur noch beweisen, dass es sich bei der derzeitigen besten Zugfolge wirklich um die bestmögliche Zugfolge handelt. Um dies zu beweisen startet man eine neue Suche mit einem ”null-Fenster”, bei dem beta beibehalten wird und alpha als beta minus 1 definiert wird. Da das Fenster klein ist, kann uns diese neue Suche nicht sagen welcher der neue beste Zug ist, man muss also, wenn die Suche einen beta Cutoff hat, wieder suchen. Sie ist aber, weil sie viel mehr Knoten ausschließt, viel schneller als die normale alpha beta Suche. Der Grund dafür, das die pv-Suche so schnell ist, das, dank der Transpositionstabelle, der beste Zug der letzten Suche als erster examiniert wird und es in der Regel ausreichend ist.

## 2.9 Iterative deepening

Wie weiß man zu welcher Tiefe das Programm suchen sollte, wenn es zum Beispiel eine Sekunde Zeit hat? Man sucht anfangs mit Tiefe eins und dann mit einer immer größer werdender Tiefe. Diese Lösung ist effektiv oft schneller als direkt zur besagten Tiefe zu suchen. Der Grund dafür ist das man wenn man sucht auch Informationen in der Transpositionstabelle und in den verschiedenen Zugordnungstabellen speichert.

## 2.10 Aspiration Windows

Der iterative deepening Algorithmus kann verbessert werden, indem die Bewertungen der vorherigen Suchen, bei der derzeitigen Suche verwendet werden. Dazu wird von der folgenden Idee ausgegangen: Die Bewertung der Stellung sollte bei der derzeitigen Suche kaum von der Bewertung der letzten Suche abweichen. Dadurch kann man anstatt mit dem normalen Suchfenster zu suchen ($\pm\infty$), mit einem reduzierten Fenster suchen (letzte Bewertung $\pm$ 125). In dem Programm wird nicht der Wert der letzten Suche für das derzeitige Suchfenster verwendet, sondern der Wert der vorletzten Suche. Der Grund dafür ist, dass die Bewertungen anfangs je nach Suchtiefe(gerade oder ungerade) oszilliert. Es ist für das Programm besser die Bewertung der Vorletzten Suche als Annäherung zu verwenden. Falls der richtige Wert der Suche sich nicht in dem Fenster befindet, gibt es ein Problem. Um dieses Problem zu mitigieren, wird bei jeder, schlecht endenden Suche das Suchfenster vergrößert. Da das Anfangsfenster klein gewählt wird, wird, um nicht zu oft zu wieder zu suchen, das Fenster exponentiell vergrößert.

## 2.11 Wurzel

Die Wurzel ist der erste Knoten in dem Suchbaum. Die Wurzel mit iterative Deepening und Aspiration Windows, entspricht folgendem Code:

```
1  int iterative_deepening(position, int search_depth){
2      bool search_pv = true;
3      int current_value = 0, best_value, delta_a = 125, delta_b = 125;
4      int window_alpha = ∞-, window_beta = ∞;
5      int last_best = 0, best_before = 0;
6      move bestmove;

8      for(int current_depth = 1; current_depth < search_depth; current_depth ++){
9          while(true){
10             if(current_depth > 2){
11                 delta_a = 125;
12                 delta_b = 125;
13                 window_alpha = best_before - delta_a;
14                 window_beta = best_before + delta_b;
15             }

17             search_result = search(current_depth, position, window_a, window_b);

19             if(search_result.fail_high){
20                 delta_b *= 2;
21                 window_beta = best_before + delta_b;
22             }
23             else if(search_result.fail_low){
24                 delta_a *= 2;
25                 window_alpha = best_before + delta_a;
26             }
27             else{
28                 break;
29             }
30         }
31         best_before = last_best;
32         last_best = search_result.score;
33         bestmove = search_result.move;
34     }
35     return bestmove;
36 }
```

# 3 Spezifische Theorie(Typ B)

## 3.1 Null move pruning

Null move pruning ist ein solcher Typ B Algorithmus, der Typ A Programmen hinzugefügt wird um sie besser Spielen zu lassen. Die Grundidee ist die schon oben beschriebene null Zug Observation, dass außer im Zugzwang, der beste Zug eine bessere Bewertung hat, als keinen Zug zu Spielen. Diese Idee machen man sich hier zunutze, indem man eine Suche mit einer reduzierten Tiefe macht ohne dabei einen Zug zu spielen. Wenn der Wert der Suche, besser als beta ist, dann kann man davon ausgehen, dass der Wert der Stellung größer als beta ist. Das Problem dabei ist zweiteilig. Erstens, wann sollte man das null move pruning erlauben? Zweitens, um wie viel sollte man die Suchtiefe reduzieren? Die Antwort auf die erste frage ist kompliziert, der Grundansatz ist aber immer derselbe. definieren erst eine Mindesttiefe bei der wir reduzieren dürfen(in diesem Fall 3). Dazu definieren wir eine Mindesttiefe bei der Wurzel, da die Suche anfangs an der Wurzel wichtig für den restlichen Suchverlauf ist(in diesem Fall auch 3). Dazu wollen wir es nicht versuchen wenn unser König in Schach ist, da die Stellung von taktischer Natur sein könnte. Wir versuchen es nur wenn die direkte Bewertung unserer Stellung größer als beta ist. Um die Probleme mit dem Zugzwang einigermaßen zu verhindern werden nur Stellungen akzeptiert mit anderen Spielfiguren als Könige und Bauern. Der Algorithmus sollte auch nur in nicht Pv-Knoten erlaubt sein, wir riskieren ansonsten ein neue primäre Variation zu übersehen. Schlussendlich wollen wir ein rekursives null move pruning erlauben(das bei der nächsten Suche auch null move pruning erlaubt wird) , nur nicht bei den direkt untergeordneten Knoten, da die Suche sonst mit unseren Zugordnungsalgorithmen interferieren könnte. Die Antwort zu der zweiten frage ist etwas einfacher, wir versuchen nicht zu stark zu reduzieren, da wir schon rekursiv reduzieren. Wir reduzieren die Tiefe um eine konstante Basis Reduktion (in diesem Fall 3). Dazu wird ein weiterer Reduktionsparameter hinzugefügt(derzeitige Suchtiefe/10). Die zwei Parameter werden beliebig, je nach Programm eingestellt, da sie von der Qualität der Bewertungsfunktion abhängig sind. Da in manchen Fällen der Unterschied zwischen beta und der direkten Bewertung groß ist muss man nicht unbedingt besonders tief suchen, da es einigermaßen sicher ist, dass die derzeitige Stellung einen beta Cutoff gibt.

## 3.2 Static null move pruning

Statisches null move pruning, auch reverse futility pruning genannt, hat das selbe Grundkonzept wie null move pruning. Es macht nur keine Suche um zu beweisen, dass die Stellung auch tatsächlich einen beta Cutoff ergibt. Es wird aber getestet, ob die derzeitige direkte Bewertung größer als beta plus ein Wert der linear von der derzeitigen Suchtiefe abhängig ist, ist. Wenn die noch verbleibende Suchtiefe größer als 7 ist, wird dieser Algorithmus nicht mehr erlaubt. Da der Algorithmus sehr aggressiv Knoten reduziert, wird er derzeit in Albatros nicht verwendet.

## 3.3 Razoring

Razoring entspricht dem statischen null move pruning, aber in dem Fall in dem die direkte Bewertung sehr schlecht ist und keine taktische Zugfolge(Quiescent search) die Bewertung verbessern kann. Es wird direkt alpha zurückgegeben. In diesem Fall ist

der, der direkten Bewertung hinzugefügte Wert auch von der Suchtiefe abhängig, aber quadratisch.

### 3.4 Late move reduction

Late move reduction ist der wichtigste Typ B Algorithmus mit dem null move pruning. Die Grundidee ist, dass die letzten Züge in der Zugliste in der Regel schlecht sind, und, dass sie daher mit einer reduzierten Tiefe gesucht werden sollten. Der Vorteil von diesem Algorithmus ist das er komplementär zum null move pruning Algorithmus ist, da null move pruning Stellungen reduziert die so gut für den derzeitigen Spieler sind, dass der andere Spieler sie nicht akzeptieren würde, und da late move reduction schlechte Züge reduziert. Damit late move reduction gut funktioniert, braucht man eine gute history heuristic, da wir je nach Platz in der Liste mehr oder weniger reduzieren. Die Größe der Reduzierungen ist abhängig von der Qualität der Bewertungsfunktion, da die Qualität der history heuristic von der Bewertungsfunktion abhängig ist. Dazu kommen zwei Fragen auf: wann sollte man reduziert suchen dürfen, und wie viel sollte man die Suchtiefe reduzieren? Erstens, sollten wir nur reduzieren dürfen, wenn die derzeitige Suchtiefe größer als 2 ist und wenn mindestens ein Zug schon gesucht wurde, die Suche wäre ansonsten schlecht. Dazu werden taktische Züge nicht reduziert, da das Programm ansonsten manche taktische Zugfolgen nicht erkennen würde. Zweitens, wird die normale Reduzierung von der folgenden Funktion bestimmt:

```
1  int reduction(int depth, int movecount, bool pv_node) {
2      if (movecount > 4 && !pv_node){
3          return (byte)((Log(depth) * Log(movecount)) / 2.75f + 0.6f);
4      } else if (movecount > 3 && pv_node)
5          return (byte)(Log(depth) * Log(movecount) / 2.75f - 0.2f);
6      } else {
7          return 0;
8      }
9  }
```

Für Züge in den der König in Schach gestellt wird und selbst einen Zug spielt um von Schach zu gehen, wird die Reduzierung verkleinert. Dazu wird bei counter- oder killer Zügen die Reduzierung auch verkleinert. Schlussendlich wird die Suchtiefe bei Zügen mit einer schlechten history Bewertung weiter reduziert und bei Zügen mit einer guten history Bewertung weniger reduziert.

## 4 Andere Algorithmen

### 4.1 Time Management

Wenn das Programm spielt sollte es je nach Zug mehr oder weniger Zeit verwenden. Dazu wird erst eine normale Zeit pro Zug berechnet. Diese Basiszeit entspricht der auf der Schachuhr noch verbliebenen Zeit durch der wahrscheinlichen Anzahl an noch verbleibenden Zügen. Wenn bei der Suche der beste Zug viel variiert, ist es eine gut Idee tiefer zu suchen, um zu versuchen einen Zug zu finden der möglichst gut ist. Und umgekehrt, wenn der beste Zug konstant immer derselbe bleibt nützt es nichts länger zu suchen, die Suchzeit kann also verkürzt werden.

## 4.2 Neuronale Netzwerke

### 4.2.1 Struktur

Neuronale Netzwerke sind spezielle Funktionen, mit beliebig erlernbaren Werten, die mit beliebig vielen Eingängen/Ausgängen versehen sind. Sie werden dadurch oft verwendet um andere Funktionen mittels Lernen anzunähern. Der Grundsatz dabei ist, dass das Netzwerk in verschiedenen Sektionen unterteilt wird. Diese Sektionen werden Schichten genannt. In jeder Schicht gibt es eine Anzahl n an sogenannten Neuronen. Dabei werden die Schichten zusammengestellt, indem jeder Neuron eine Verbindung mit allen Neuronen(z.B. der Schicht l - 1) der nächsten Schicht (l) hat. Diese Verbindungen zwischen zwei Neuronen werden Gewichte genannt, da sie den Ausgang des letzten Neurons proportional zu ihrem Wert ändern. Die Werte der Gewichte sind erlernbar. Jeder Neuron der Nächsten Schicht (l), bekommt als Eingabe die Summe aller gewichteten Ausgaben der Neuronen der letzten Schicht(l - 1). Zu dieser Eingabe wird ein ebenfalls erlernter Schwellenwert addiert. Dieser neue Wert wird als Eingang der nicht linearen Aktivierungsfunktion verwendet. Die Aktivierungsfunktion wird gebraucht, da ansonsten alle Verbindungen zwischen allen Schichten des Netzwerkes zu einer einzigen Schicht vereinfacht werden könnten. Ein weiteres Problem ist, dass ohne Aktivierungsfuntkion das Netzwerk nur linear sein kann. Historisch wurden für Aktivierungsfunktionen Sigmoide verwendet, Heutzutage bevorzugt man eher sogenannte ReLUs(Rectified Linear Unit). Sie werden wie folgt definiert:

$$ReLU(x) = \begin{cases} x & (x > 0) \\ 0 & \text{ansonsten} \end{cases}$$

Um die Ansicht etwas zu vereinfachen kann man die Gewichte zwischen zwei Schichten als Matrix darstellen, mit der Größe => Anzahl der Neuronen in der Schicht l x Anzahl an Neuronen in l + 1. Die Schwellenwerte kann man als Vektoren darstellen. Dann gilt für den Ausgangsvektor der Schicht l:

$$\vec{y} = \vec{z_l} + \vec{b_l}$$
$$\vec{z_l} = \vec{o_{l-1}} \boldsymbol{W}_l$$
$$\vec{o_l} = \begin{cases} \vec{x} & \text{falls es die erste Schicht ist} \\ \sigma(\vec{z_l} + \vec{b_l}) & \text{ansonsten} \end{cases}$$

wobei:

$\vec{x}$ = Eingang des Netzwerkes
$\vec{y}$ = Ausgang de Netzwerkes
$\vec{z}$ = Summe der gewichteten Ausgaben
$\vec{o}$ = Ausgang von einer Schicht
$\boldsymbol{W}$ = Gewichtsmatrix
$\sigma$ = Aktivierungsfunktion

### 4.2.2 Der Lernmechanismus(Backpropagation)

Die Backpropagation ist der Standardalgorithmus um ein neuronales Netzwerk lernen zu lassen. Die Grundidee dabei ist, dass das neuronale Netzwerk versucht anhand eines Eingangs eine Vorhersage zu treffen(die Vorhersage wäre in diesem Fall $\vec{y}$). Neben der Vorhersage haben wir mit der Trainingsprobe auch den Wert, den der Ausgang je nach Eingang haben sollte $\vec{a}$. Wir können also einen Fehler zwischen Vorhersage und richtigen Wert berechnen. Dieser Fehler wird anhand der sogenannten Kostenfunktion berechnet. Die Kostenfunktion entspricht dem Durchschnitt aller Fehler in allen Ausgängen $y_n$ zum Quadrat:

$$C(\vec{a}, \vec{y}) = \frac{1}{m} \sum_{n=1}^{m} (a_n - y_n)^2$$

Unser Grundsatz ist, dass wir versuchen $C(\vec{a}, \vec{y})$ für alle Testproben die wir besitzen zu minimieren. Mann kann sich also vorstellen, dass der Durchschnitt von $C(\vec{a}, \vec{y})$ in den Testproben, in Abhängigkeit von den erlernbaren Parametern als eine n-dimensionale Ebene darstellbar ist. Wir versuchen, mithilfe von Backpropagation die trainierbaren Parameter so zu verändern, dass wir uns in Richtung eines lokalen Minimums, der Kostenfunktion im Durchschnitt über alle Testproben bewegen. Das wird durch Differenziation der Kostenfunktion nach den Gewichten erreicht. Um das Endergebnis zu erreichen, werden wir die Funktionsweise eines einzelnen Neurons j noch etwas genauer anschauen:

$$\vec{x} = \vec{o}_{l-1}$$
$$z_j = \sum_{i=1}^{n} x_i \omega_{ij}$$
$$o_j = \sigma(z_j)$$

wobei:

$\vec{x}$ = der Eingang des Neurons j
$\omega_{ij}$ = das Gewicht zwischen der Eingabe i und dem Neuron j
$o_j$ = der Ausgang des Neurons j

Auf einen Schwellenwert wird hier verzichtet, da er als Gewicht, das mit einem Neuron mit einem konstanten Ausgangswert 1 verbunden ist, gesehen werden kann.

Die partielle Ableitung der Kostenfunktion $C$ nach dem Gewicht $\omega_{ij}$ ergibt sich unter Verwendung der Kettenregel:

$$\frac{\partial C}{\partial \omega_{ij}} = \frac{\partial C}{\partial o_j} \frac{\partial o_j}{\partial z_j} \frac{\partial z_j}{\partial \omega_{ij}}$$

Man kann sich diese Ableitung vorstellen, als wie eine kleine Veränderung von dem Gewicht $\omega_{ij}$, den Wert der Kostenfunktion $C$ beeinflusst. Jetzt können wir jedes Element der Ableitung separat betrachten. Es gilt somit:

$$\frac{\partial o_j}{\partial z_j} = \sigma'(z_j)$$

$$\frac{\partial z_j}{\partial \omega_{ij}} = o_i$$

$$\delta_j = \frac{\partial C}{\partial o_j}\frac{\partial o_j}{\partial z_j} = \begin{cases} \sigma'(z_j)2(a_n - y_n) & \text{falls j die letzte Schicht ist} \\ \sigma'(z_j)\sum_k \delta_k \omega_{jk} & \text{ansonsten} \end{cases}$$

$$\Delta\omega_{ij} = -\eta\delta_j o_i$$

wobei:

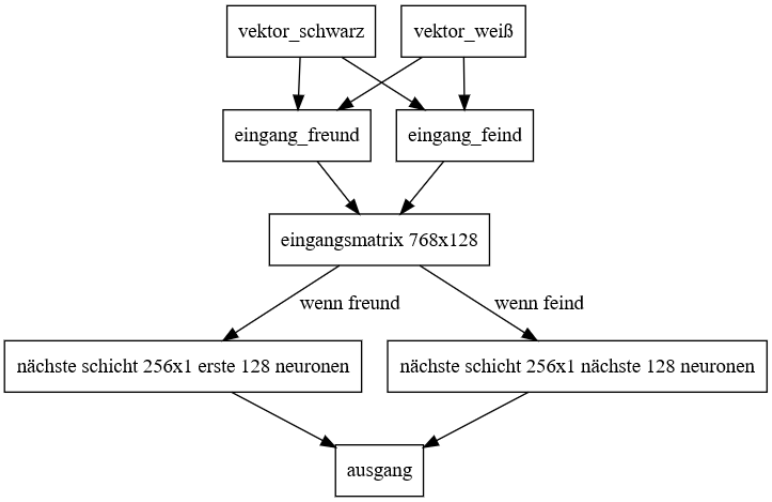
$\delta_j$ = der Fehler des Neurons j
$\Delta\omega_{ij}$ = die Änderung des Gewichts $\omega_{ij}$ von Neuron i zu j
$\eta$ = die feste Lernrate, die die Lerngeschwindigkeit beeinflusst
$k$ = der Index der nachfolgenden Neuronen von j

Diese Lernart wird überwachtes Lernen genannt, da der Wert, den der Ausgang, je nach Eingang, haben sollte uns schon bekannt sind.

### 4.2.3 NNUE Neuronale Netzwerke

NNUE (Efficiently Updatable Neural Network) ist eine bestimmte Art von neuronalen Netzwerken, die speziell als Bewertungsfunktion, für das Spiel Shogi entwickelt wurden. Da Shogi eine Variante des Schachspiels ist wurden die Netzwerke schnell für Schachprogramme adaptiert. Der Grundsatz von diesen Netzwerken ist, dass sie möglichst groß werden ohne zu langsam zu sein, während sie auf der CPU berechnet werden. Das wird mithilfe folgender Techniken erreicht:

Erstens, ist die Grundidee, dass die Berechnungen in der ersten Schicht einfach sein sollten, der Eingangsvektor sollte also nur Einsen und Nullen beinhalten. Dazu sollte er, wenn möglich eine große Vielzahl an Nullen beinhalten. Die Darstellung der Stellung als Vektor ist also entscheidend. Die Idee der Darstellung in dem Programm ist, dass alle Spielfiguren nach Typ und nach Platz unterteilt werden. Da es zwölf verschiede Figurtypen und 64 verschiedene Plätze auf dem Schachbrett gibt, beinhält der derzeitige Eingangsvektor $12 * 64 = 768$ Dimensionen. Dazu muss das Netzwerk aber auch noch Informationen über welche Seite gerade spielen darf bekommen. Dazu wird der Eingangsvektor verdoppelt. Beide dieser neuen Eingangsvektoren beinhalten dieselbe Information, sind aber aus der Perspektive ihres Spielers dargestellt. Es wird dann je nach Runde, zum Beispiel der Vektor mit der Perspektive von Weiß entweder in den platz von Freund, oder Feind gestellt. Das selbe gilt für den Vektor von Schwarz. Der Grund dafür, dass wir für den Eingang nur Einsen und Nullen wollen ist, dass der Ausgang der ersten Schicht besonders einfach zu berechnen ist. Wir müssen, anstatt einer normalen Vektor Matrix Multiplikation, bei der alle Werte berechnet werden müssen, nur die teile der Matrix, bei denen der Eingangsvektor den Wert 1 hat zum Gesamtprodukt addieren. Wir haben dazu ein weiteren Vorteil, der ist, dass wir den Ausgang der ersten Matrix inkrementell, also je nach Zug, auffrischen können. Damit das richtig funktioniert verwenden wir anstatt einer einzelnen Eingangsmatrix mit der vollen Größe des Eingangs, eine Eingangsmatrix mit der Hälfte der Größe des Eingangs. Die Eingangsmatrix wird also 2 mal verwendet einmal für Freund und einmal für Feind. Die Topologie des Netzwerkes entspricht folgendem Bild:




Zweitens, da das Netzwerk auf der CPU berechnet wird, verwenden wir die Avx2 Erweiterung des x86 Befehlssatzes. Der Vorteil von Avx2 ist, dass wir Berechnungen mit 256-bit Vektoren, mit 8, 16 und 32 Bit ganze Zahlen machen können. Das einzige Problem dabei ist, dass die Vektoren und Matrizen in unserem Netz eigentlich Kommazahlen sind. Die Lösung dazu heißt Quantisierung, die Werte der Gewichte haben je nach Typ eine andere Auflösung. Sie haben ein kleines Problem, da sie nicht so genau wie Kommazahlen sind und Maximal- /Minimalwerte haben. Da die Werte in dem Netzwerk sehr stark quantisiert sind, wird als Aktivierungsfunktion die clipped ReLU Funktion verwendet:

$$CReLU(x) = \begin{cases} 0 & (x < 0) \\ 1 & (x > 1) \\ x & ansonsten \end{cases}$$

In Albatros ist das Netzwerk implementiert, getestet, ist aber noch am Anfang des Lernprozesses.

# 5 UCI Programm Albatros

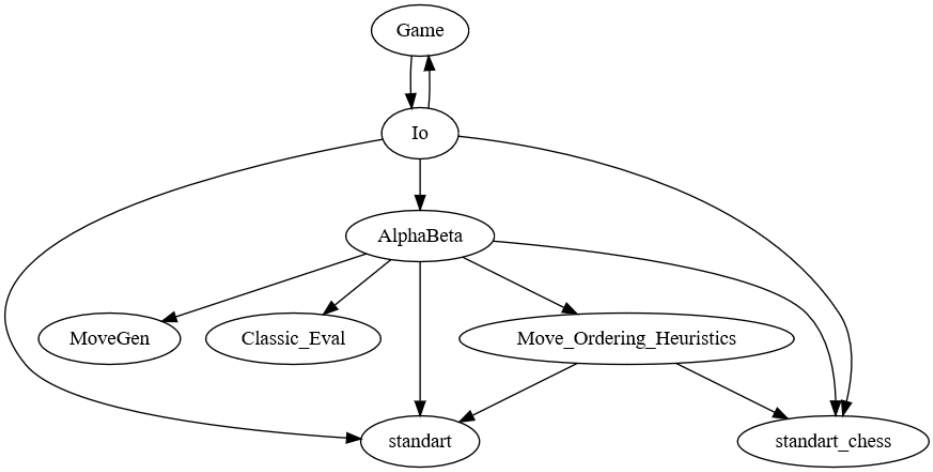
Das Schachprogramm Albatros wurde dazu konzipiert um oben beschriebene Algorithmen umzusetzen. Das Programm wurde komplett in der Programmiersprache C# geschrieben. C# ist ziemlich untypisch für Schachprogramme. C# hat Garbage Collector und kompiliert zu einer virtuellen Maschine. C, C++ und Rust gelten als angemessener. Die Sprache ist aber doch gut geeignet, da sie direkten Zugang zur Hardware erlaubt und da im ersten Schritt die Algorithmen wichtig sind.

## 5.1 UCI

UCI ist ein universelles Textprotokoll für Schachprogramme. Fast jedes Schachprogramm heutzutage verwendet es. Es wird von den graphischen Benutzeroberflächen

verwendet um mit Schachprogrammen zu interagieren. Das Protokoll hat ein paar Besonderheiten, da zwei bestimmte Kommandos zu jeder Zeit funktionieren sollten, selbst wenn das Programm schon sucht. Die Lösung dafür, war ein zweiter Thread, der die Kommandos überwacht, und den Berechnungsthread steuert.

## 5.2 generelle Architektur




Die Graphik zeigt die Abhängigkeiten, von jedem Modul zu anderen Modulen. Game ist der Startpunkt des Programms. Io interpretiert die UCI Kommandos und interagiert mit der Alpha-Beta Suche. Die Alpha-Beta Suche benötigt die Bewertung(Classic_Eval), die Methoden zur Ordnung der Züge(Move_Ordering_Heuristics) und das Modul zur Generierung von legalen Zügen(MoveGen). Die Zuggenerierung in Albatros, erfolgt im Vergleich zu anderen Schachprogrammen, ziemlich langsam. Das liegt daran, dass zur Zuggenerierung es eine spezifische theoretische Weise gibt, die die Züge sehr Schnell generiert, die in dem Programm nicht verwendet wurde. Die Zug Generierung wurde in dem Programm ohne Theorie programmiert. Das Modul standart hat Funktionen, die bei der Entwickelung von Albatros oft gebraucht wurden, aber in dem Standard C# nicht zu finden waren. Das Modul standart_chess erfüllt denselben Zweck wie standart, außer das standart_chess nur Standardfunktionen für Schach hat.

## 5.3 Trainieren des neuronalen Netzwerkes

Das Trainieren des neuronalen Netzwerks wird in zwei verschiedenen Schritten umgesetzt. Es wird erst eine große Anzahl an Spielen von Albatros generiert. Die Spiele werden mit Selbstspiel gespielt, das Programm spielt also für beide Seiten mit der selben Suchtiefe. Die Suchtiefe ist in diesem Fall 1, da eine tiefe Suche in diesem Kontext nicht so wichtig ist, wie eine große Masse an Spielen. Die Startstellung von jedem dieser Selbstspiele wird nach einer zufälligen Anzahl(zwischen 10 und 20) an zufälligen Zügen erreicht. Da wir möglichst schnell, möglichst viele Spiele spielen wollen, ist dieses Stück von dem Programm mit mehreren Threads programmiert worden. Nach der Generierung der Spiele, lernt das neuronale Netzwerk den Wert der Endstellung des Spieles vorherzusagen.

# 6 Tests und Ergebnisse

## 6.1 Elo

Der ELO Algorithmus wird verwendet um zu berechnen mit welcher Wahrscheinlichkeit ein Spieler A gegen einen Spieler B gewinnt. Jeder Spieler hat eine Zahl, die ELO Zahl, die seine Spielstärke misst. Die Wahrscheinlichkeit, dass A gegen B gewinnt wird mit folgender Formel berechnet:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

wobei:

$E_A$ = Die Wahrscheinlichkeit, dass A gewinnt
$R_A$ = ELO von A
$R_B$ = ELO von B

Die 400 in der Formel sind empirisch. Die Veränderung von $R_A$, nach einem Spiel entspricht:

$$R'_A = k(S_{AB} - E_A)$$

wobei:

$k$ = Eine beliebige Konstante, die dem Entwickelungsstandart in dem Spiel entspricht
$S_{AB}$ = Wie Das Spiel zwischen A und B ausgefallen ist

## 6.2 Tests

Tests sind im Kontext von Schachprogramme ziemlich komplex. Wenn wir nicht wissen was der beste Zug in einer Stellung ist, wie wissen wir ob unser Programm gut spielt oder nicht? Die Lösung heißt sehr viele Spiele gegen möglichst vielen anderen Programm einer ähnlichen Spielstärke spielen lassen. Und genügend Spiele spielen um sicher zu gehen, dass die derzeitige Version des Programmes besser spielt als die alte.

Der derzeitige ELO Wert, des Programmes entspricht im etwa 1800, was für Menschen ziemlich gut ist, aber immer noch sehr weit von gesuchten übermenschlichen Niveau(ungefähr 3000 ELO) ist.

# 7 Fazit

## 7.1 Zusammenfassung

In diesem Dokument, haben wir gesehen wie moderne Schachprogramme funktionieren. Wir haben gesehen welche Algorithmen verwendet werden, wieso sie verwendet werden und wie sie funktionieren. Leider haben wir das Ziel des Programmes noch nicht ganz erreicht. Das Programm ist Open Source und auf Github zu finden. Falls sie gegen das Programm spielen wollen, es ist auf der Schachwebseite Lichess zu finden.

## 7.2 Weitere Arbeit

Es wäre nützlich die Algorithmen besser abzustimmen. Dazu gibt es noch weitere Verbesserungsmöglichkeiten, für die Bewertung und für die Suche. Es gibt auch noch weitere Algorithmen, die man hinzufügen könnte. Es wäre auch eine gute Idee das Programm in der Programmiersprache Rust umzuschreiben. Das neuronale Netzwerk sollte weiter lernen. Dazu gibt es ein paar Ideen, die noch nie implementiert wurden.

# 8 Glossar

## 8.1 Alpha

Der beste Wert des derzeitigen Spielers

## 8.2 Beta

Der beste Wert des anderen Spielers

## 8.3 Alpha Cutoff

Wird auch fail-low genannt. Tritt in der Alpha Beta und der Pv-Suche auf, wenn in einem Knoten kein neuer bester Wert gefunden werden konnte. Diese Knoten werden in der Pv-Suche All-Knoten genannt.

## 8.4 Beta Cutoff

Wird auch fail-high genannt. Tritt in der Alpha Beta und der Pv-Suche auf, wenn in einem Knoten der derzeitige beste Wert alpha so gut ist, dass man in diesem Knoten nicht mehr weitersuchen muss. Diese Knoten werden in der Pv-Suche Cut-Knoten genannt.

## 8.5 Suchfenster

Das Suchfenster ist der Wertebereich zwischen Alpha und Beta. Wenn ein Wert außerhalb des Suchfensters ist, dann erfolgt ein Cutoff.

## 8.6 Null-Fenster

Wenn das Suchfenster ein sogenanntes Null-Fenster ist, dann ist die Differenz zwischen Alpha und Beta die Kleinstmögliche Differenz, die nicht null ist.

## 8.7 Null Zug Observierung

Wenn ein Spieler passt, und den zweiten Spieler einen weiteren Zug spielen lässt, dann ist die Bewertung dieses Null Zuges in der Regel schlechter als bei dem besten Zug in der Stellung.

# 9 Quellen

## 9.1 Hauptrecource

https://www.chessprogramming.org/Main_Page

## 9.2 Lernalgorithmen für neuronale Netzwerke

https://de.wikipedia.org/wiki/Backpropagation
https://de.wikipedia.org/wiki/%C3%9Cberwachtes_Lernen

## 9.3 NNUE neuronale Netzwerke

https://github.com/asdfjkl/nnue/blob/main/nnue_en.pdf
https://github.com/glinscott/nnue-pytorch/blob/master/docs/nnue.md#quantization

## 9.4 Starke Open Source Schachprogramme

https://github.com/official-stockfish/Stockfish
https://github.com/AndyGrant/Ethereal

## 9.5 Weitere Inspiration

https://arxiv.org/abs/1712.01815
https://arxiv.org/abs/1902.10565
https://www.sciencedirect.com/science/article/pii/S0004370201001291
https://www.ml.informatik.tu-darmstadt.de/papers/czech2019deep.pdf