Erweiterung einer Schach-Engine zu einer Team-Schachvariante

Bachelorarbeit im Fachgebiet Elektrotechnik/Informatik
der Universität Kassel

vorgelegt von

Nikolas Luke
geb. am 20. März 1981 in Uslar

Eingereicht am 11. Mai 2011

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

angefertigt beim
Fachbereich Programmiersprachen/-methodik
Wilhelmshöher Allee 73
34125 Kassel

Inhaltsverzeichnis

Einleitung

Im Jahr 1957 war man der Ansicht, dass Computer *künstliche Intelligenz* besitzen müssten, um sehr gut Schach spielen zu können. Das Schachspiel stellte in diesem Zusammenhang eine Aufgabe dar, "zu deren Lösung Intelligenz notwendig ist, wenn sie vom Menschen durchgeführt wird" [1]. Heute weiß man, dass Schach als mathematisches Spiel von einem Computer mit einem festen Algorithmus hinreichend gut ausgerechnet werden kann und dabei bereits Großmeisterniveau erreicht wird.

Die vorliegende Bachelorarbeit beschäftigt sich neben der Schachprogrammierung mit dem Spiel *Bughouse*. Bughouse ist eine von vielen Schachvariationen, die zwar mit denselben Figuren wie im normalen Schach auf Schachbrettern gespielt werden, deren Regeln sich aber von Schach unterscheiden. Bughouse stellt dabei eine originelle Variation dar. In diesem Teamspiel, welches in Deutschland mit leicht modifizierten Regeln als Tandem-Schach bekannt ist, wird an zwei nebeneinander liegenden Brettern parallel nach weitgehend normalen Schachregeln gespielt. Die Teams sitzen sich dabei gegenüber und die Teampartner erhalten unterschiedliche Farben, d.h. die beiden Schachbretter sind wie in Abbildung 1 gegeneinander gedreht.

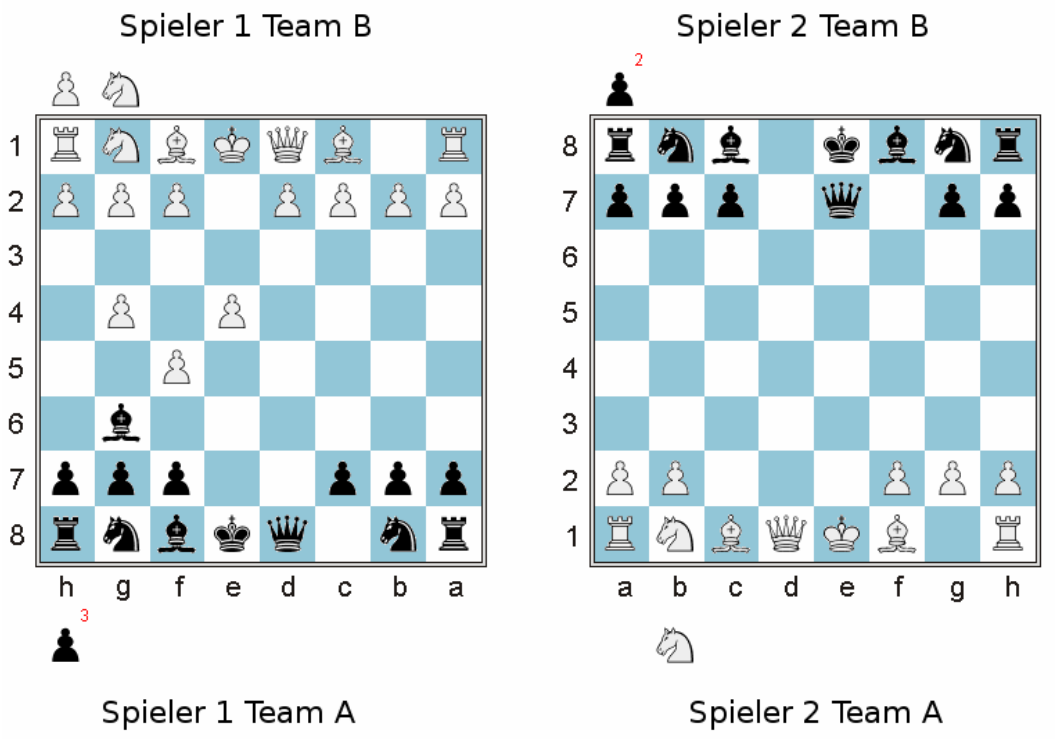


Abbildung 1 Laufende Bughouse Partie

Figuren, die ein Spieler während des Spiels schlägt, werden an den Partner weitergegeben, der diese als eigene Figuren statt eines regulären Schachzugs auf ein beliebiges Feld seines Brettes stellen kann. Dabei dürfen diese Figuren auch direkt Schach oder Matt bieten. Hat ein Spieler durch Mattsetzen oder Ablauf der Uhr des Gegners gewonnen, hat damit auch sein Team gewonnen. Die Partie am anderen Brett wird abgebrochen.

Eine Bughouse-Partie ohne den Einsatz von Schachuhren ist kaum denkbar, da die Spieler dazu neigen auf neue vom Partner geschlagene Figuren zu warten, was mit "unendlich Zeit" schnell zu einem Erliegen des Spielflusses führt. Zusammen mit dem Umstand, dass die geschlagenen Figuren nur die Besitzer wechseln und damit nicht aus dem Spiel verschwinden, ist eine solche Bughouse-Partie in einem angemessenen Zeitrahmen eher unwahrscheinlich. Gerade durch den Zeitfaktor gestalten sich die Partien schnell, überraschend und mitunter für Spieler und Zuschauer sehr unterhaltsam, da Fehler unter Zeitdruck, angesichts der im Gegensatz zum normalen Schach enorm gestiegenen Anzahl der Zugmöglichkeiten, geradezu vorprogrammiert sind.

Bughouse ist eine eher seltene Schachvariante und es existieren nur wenige Schach-Engines, die es beherrschen. Hinzu kommt, dass eine angepasste GUI mit zwei Brettern benötigt wird und GUIs gewöhnlich nicht speziell für dieses Spiel angepasst sind. Nach eigenen Recherchen gibt es daher kaum Möglichkeiten eine Bughouse-Partie lokal auf einem Computer zu spielen. Auf Schachservern laufende Bughouseengines haben zudem die Eigenart, sofort, ohne Zeitverlust einen Antwortzug auszuführen. Der Spielspaß der dabei entsteht, stärker als ein Computer spielen zu können, wird durch die Zuggeschwindigkeit, mit der der Computer dies kompensiert, etwas getrübt.

Im Rahmen dieser Bachelorarbeit wurde eine eigene Bughouse-Engine auf Basis der Open Source Java Schach-Engine *Mediocre* (nebst eigener GUI) entwickelt. Da die Schach-Engine *Mediocre* viele moderne Schachprogrammierungstechnologien einsetzt, interessante Datenstrukturen besitzt und eine überdurchschnittliche Spielstärke aufweist, wurde sie als Basis gewählt. Die dabei entstandene Bughouse-Engine ist auch für mit dem Spiel vertraute Schachspieler ein interessanter Gegner. Damit stellt das praktische Ergebnis dieser Bachelorarbeit eine sinnvolle Ergänzung zum Angebot der Schachserver dar.

Während in der Informatik das Interesse am Computerschach zurückgegangen ist, bleibt die Programmierung von Bughouse-Engines als seltenes, kaum erforschtes Gebiet, noch eine Herausforderung. Die stark erhöhte Zahl an verschiedenen Zugmöglichkeiten wirft die Frage

auf, ob mit den bekannten Techniken der Schachprogrammierung, immer noch gute Züge berechnet werden können und der Computer für Menschen auch in der Schachvariation ein ernstzunehmender Gegner bleibt. Diese Frage kann durch Beobachtung der Server-Engines nicht beantwortet werden, da diese nicht den Anspruch haben, starke Züge zu spielen, weil das schnelle Ziehen gewöhnlich ausreicht, menschliche Spieler in Schwierigkeiten zu bringen.

Im ersten Kapitel dieser Ausarbeitung wird zunächst allgemein auf die Programmierung von Schachcomputern und deren GUIs (Graphical User Interface = Grafische Benutzeroberfläche) eingegangen. Anschließend werden die grundlegenden Algorithmen der Schachprogrammierung beschrieben sowie eine Auswahl von fortgeschrittenen Algorithmen behandelt. Das erste Kapitel schließt mit einer kurzen Vorstellung der führenden Schach-Engines und deren bisherigen Erfolgen. Das Open Source Programm *Mediocre* wird im zweiten Kapitel auf die darin eingesetzten Technologien und des verwendete Datenmodell untersucht sowie dessen Stärken und Schwächen beschrieben. In Kapitel 3 wird auf die Programmierung der Schachvariation "Bughouse" eingegangen und Unterschiede zur Programmierung von normalem Schach aufgezeigt. Anschließend wird beschrieben, wie Mediocre zur Schachvariation Bughouse umgeschrieben und die neue Engine getestet wurde. Im vierten Kapitel wird der Inhalt dieser Bachelorarbeit noch einmal zusammengefasst, die Ergebnisse beurteilt und ein Fazit gezogen.

1. Moderne Schach-Engines und deren Basistechnologien

1.1. GUIs und Engines

Die heutigen Schachprogramme bestehen gewöhnlich aus zwei Teilen. Die Schach-Engine, welche im engeren Sinne als Schachcomputer bezeichnet wird, bekommt eine Schachstellung vorgegeben und gibt nach einer vorgegebenen Zeit oder Suchtiefe den als besten befundenen Zug zurück.

Mit der grafischen Benutzerschnittstelle (GUI) können Partien gegen den Computer gespielt oder Onlinepartien gegen Menschen ausgetragen werden. Bei einer Partie gegen eine Schach-Engine wird diese von der GUI nach dem Computerzug befragt. Die GUI entscheidet auch, ob ein Spiel "auf Zeit" oder durch Matt gewonnen worden ist. Je nach Ausstattung können über die GUI auch Partiedatenbanken, Eröffnungsbücher und Zugstatistiken zur Analyse und zum Training herangezogen werden.

Die Kommunikation zwischen GUI und Engine wird am häufigsten mit dem *Chess Engine Communication Protocol* (CECP), das auch als XBoard Protokoll bekannt ist oder dem neueren *Universal Chess Interface* (UCI) durchgeführt. Diese stellen einen Quasi-Standard dar. In beiden Fällen kommunizieren GUI und Engine über Konsolenein- und ausgaben miteinander. Das UCI Protokoll sieht für den Anwender eine beträchtliche Anzahl an Features vor, welche die Engine unterstützen muss. Dabei können beispielsweise Schachstellungen bis zu einer vorgegebene Tiefe, einem Zeitlimit oder sogar auf unbegrenzte Zeit (d.h. bis zum manuellen Abbruch durch den Benutzer) analysiert werden.

Die beliebtesten Schach-Engines sind meist kommerziell und werden teilweise mit den gleichen GUIs angeboten.

1.2. Basisalgorithmen

Im Jahr 1950 veröffentlichte der Mathematiker Claude Shannon (1916 - 2001) den MiniMax Algorithmus, welcher die Schachprogrammierung noch heute beeinflusst. Dieser im folgenden beschriebene Algorithmus ist in praktisch jeder Schachengine zu finden.

Shannon glaubte, dass Computer mit seinem Algorithmus (auch unter Berücksichtigung der steigenden Rechenstärke) durch den großen Suchraum bestenfalls mittelmäßiges Schach spielen können. Der Algorithmus konnte aber durch eine entscheidende Modifikation zum im anschließend beschriebenen "Alpha-Beta-Pruning" Algorithmus erweitert werden, so dass die heutigen Schach-Engines enorme Spielstärken erreichen [2].

1.2.1. MiniMax Algorithmus

In normalen Schachpositionen existieren etwa 20 bis 40 legale Züge, die in einem aufzustellenden Spielbaum auf ihren Wert geprüft werden müssen. Ähnlich wie ein Mensch muss ein Computer dafür auch alle Gegenzüge prüfen und deren Gegenzüge usw. Shannon fand heraus, dass dazu nicht viel mehr als ein Zuggenerator, eine Suchfunktion und eine Stellungsbewertungsfunktion nötig sind [2].

Der von ihm vorgestellte Algorithmus geht davon aus, dass beide Spieler ihre besten gefundenen Züge ziehen werden und keiner der Spieler Züge übersieht. Dadurch soll der objektiv beste Zug gefunden werden.

Bei Nutzung des MiniMax Algorithmus wird der Spielbaum bis zu einer vorgegebenen Tiefe durch Ausführen der möglichen Züge aufgebaut. Der Zuggenerator erstellt dabei eine Liste legaler Züge, welche in einer Spielsituation möglich sind. Dabei werden den Endstellungen an den Blättern Werte durch eine Bewertungsfunktion zugeordnet. Positive Zahlen geben dabei einen Vorteil für den Spieler A an, negative für Spieler B. Bei einem Wert von Null liegt eine ausgeglichene Stellung vor oder aber eine Stellung in der keine Partei mehr gewinnen kann. Die Wertungszahlen können beliebig skaliert werden. Eine gewonnene Stellung erhält dabei die höchste Bewertung. Diese Werte werden an die Vorgängerknoten zurückgegeben. In jedem Knoten, bei dem Spieler A am Zug ist, wird der höchste gefundene Wert zurück-

gegeben während bei Zügen, bei denen Spieler B am Zug ist, der niedrigste gefundene Wert zurückgegeben wird.

In Abbildung 2 ist der Ablauf grafisch bis zu der vorgegebenen Tiefe von 4 dargestellt. Der maximale Knotengrad ist dabei zur besseren Übersicht auf zwei beschränkt. Man beachte, dass vorerst der linke Baumzweig bis in Tiefe 4 aufgebaut wird und dort die erste Bewertung stattfindet. Spieler B, der in Tiefe 3 am Zug ist, bekommt aus Tiefe 4 die Rückgabewerte 10 und +∞. Da kleinere Werte für Spieler B besser sind, würde sich dieser für den niedrigeren Wert entscheiden. Also wird die Zahl 10 zurückgegeben. Nach Erforschung aller Varianten wird der an der Wurzel von Spieler A gespielte Zug eine Bewertung erhalten, welche mit weiteren für Spieler A noch zu bewertenden Zügen verglichen werden muss.

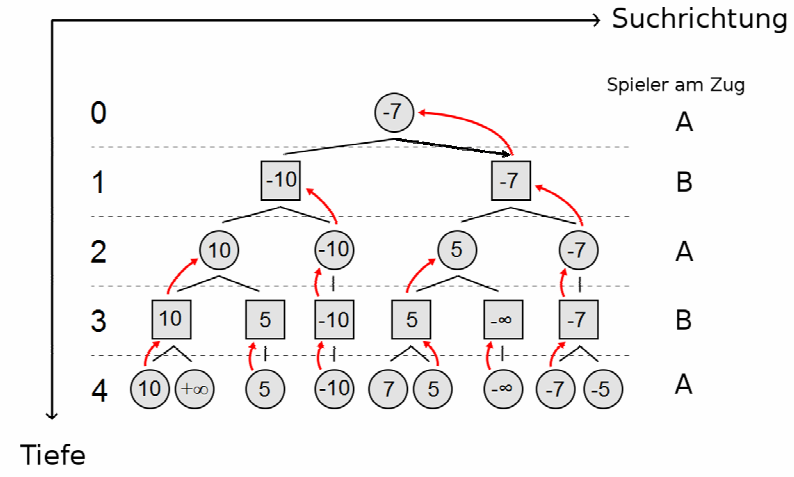


Abbildung 2 Baumsuche nach dem MiniMax Prinzip vgl. [4]

Die Implementierung des MiniMax Algorithmus sieht zwei Methoden vor, welche sich gegenseitig aufrufen und so den Spielverlauf simulieren. Während die eine Methode sich analog zu Spieler A für den maximalen Rückgabewert der rekursiven Aufrufe entscheidet, gibt die zweite Methode den minimalen Rückgabewert an den Aufrufer zurück. Zur Vereinfachung werden die beiden Methoden gewöhnlich unter Zuhilfenahme eines Flags oder, wie in Abbildung 3, durch eine Negation des Methodenaufrufs zu einer vereint. Man spricht bei einer solchen Vorgehensweise von einer NegaMax Formulierung eines Algorithmus. Man beachte, dass die Funktion in Abbildung 3 mit der maximalen Tiefe aufgerufen wird und diese sich bei jedem Aufruf verringert, bis in einer noch verbleibenden Tiefe von 0 die Blätter des Baumes erreicht sind.

```
int negaMax(int depthleft) {
    if (depthleft == 0) return evaluate(); // Bewertungsfunktion an den Blättern

    int max = +INFINITY;
    for (all moves) {                      // Simulieren aller Zuege
        doNextMove();
        score = -negaMax(depthleft-1);     // Suche rekursiv bis zu den Blättern...
        if (score > max)                   // Finde den besten Wert fuer
            max = score;                   // den aktuellen Spieler
        undoMove();
    }

    return max;                            // gebe den besten Wert zurueck
}
```

Abbildung 3 NegaMax Pseudocode vgl. [5]

1.2.2. Alpha-Beta-Suche

Der Alpha-Beta Algorithmus (Alpha-Beta-Pruning) stellt eine entscheidende Verbesserung zum MiniMax Suchalgorithmus dar, indem er bestimmte Zweige im Suchbaum nicht weiter verfolgt. Es handelt sich dabei um einen *branch-and-bound* Algorithmus. Die Überlegung ist folgende: Wenn Spieler A bereits einen relativ guten Zug gefunden hat und eine Alternative untersucht, genügt bereits ein starker Zug des Gegners, um diesen Alternativzug zu verwerfen. Es ist nicht mehr wichtig, ob der Gegner noch weitere vielleicht stärkere Züge machen kann. Spieler A würde sich ohnehin für seinen bereits gefundenen starken Zug entscheiden.

Sehen wir uns folgendes Beispiel nach "CHESS Programming WIKI"[6] an:
Sagen wir Weiß ist am Zug und wir suchen bis in eine Tiefe von 2 (das bedeutet wir betrachten alle weißen Züge und deren schwarzen Antwortzüge und bewerten dann die Endstellungen). Nehmen wir einen möglichen Zug von Weiß und nennen ihn Zug #1. Nach der Analyse aller schwarzen Antworten und Bewertung aller Endstellungen stellen wir fest, dass Zug #1 eine Bewertung von 0 erhält, da für keine der beiden Spieler Vorteile bestehen.
Nun fahren wir fort und sehen uns andere Züge von Weiß an. So auch den Zug #2. Wir stellen bei diesem Zug fest, dass schon der erste Antwortzug von Schwarz einen Turm gewinnen würde! In dieser Situation können wir alle weiteren Züge von Schwarz ignorieren, da wir schon jetzt wissen, dass Zug #1 besser ist. Es ist nicht mehr interessant, ob Zug #2 bei Betrachtung aller schwarzen Antwortzüge noch schlechter als bereits angenommen ausfällt, da wir wissen, dass wir mit Zug #1 wenigstens eine ausgeglichene Stellung erhalten.

Mit steigender Suchtiefe haben beide Spieler Möglichkeiten, aufgrund ihrer bereits berechneten Züge Alternativzüge zu verwerfen. Im Suchbaum übergibt jeder Knoten einen Wert Alpha an seine Kinder, welcher die Bewertung des für Spieler A bereits gefundenen besten Zugs darstellt. Dazu kommt ein Wert Beta, welcher den besten für Spieler B gefundenen Zug repräsentiert. Es werden also nur noch Züge weiter untersucht, welche noch nicht unter die untere Grenze des Alpha Wertes gefallen sind und die Obergrenze des Wertes Beta noch nicht überschritten haben. Wird von Spieler A die Untersuchung von alternativen Zügen weggelassen spricht man von einem *Alpha-Cut* (oder auch *Alpha-Cutoff*). Verwirft Spieler B das Untersuchen von Zügen wird das Beta-Cut genannt. In Abbildung 4 brauchten die grauen Baumzweige nicht weiter untersucht werden. Die gelben Pfeile zeigen, wo ein Rückgabewert als Alpha oder Beta Wert später in alternativen Varianten mitgeführt wird.

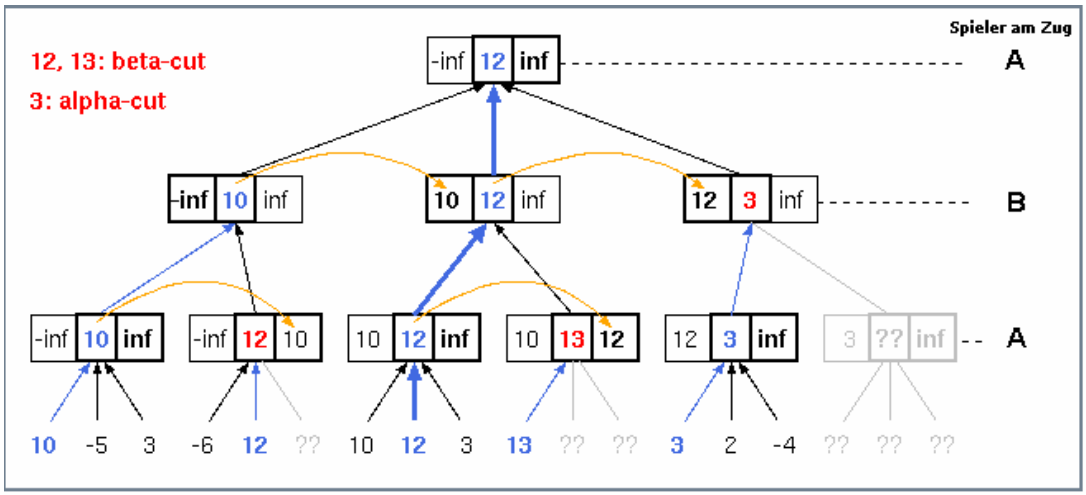


Abbildung 4 Beispiel Alpha-Beta Suche vgl [7]

```
int alphaBeta(int alpha, int beta, int depthleft) {
   if (depthleft == 0) return evaluate();           // Bewertungsfunktion an den Blättern

   for (all moves) {                                // Simulieren aller Zuege
      doNextMove();
      if (it is playerAs turn) {
        a = max( a, alphaBeta(a, b, depthleft-1) );
        if (a >= b ) return a;  // Beta-Cut:
                                // Spieler A hat gerade einen guten Zug mit der Bewertung a gefunden.
                                // Spieler B wird seinen Zug davor deshalb nicht ziehen, da ihm
                                // schon ein besserer Alternativzug mit der Bewertung b bekannt ist
      }
      else {
        b = min( b, alphaBeta(a, b, depthleft-1) );
        if (a >= b ) return b;  // Alpha-Cut:
                                // Spieler A hat gerade einen guten Zug mit der Bewertung a gefunden.
                                // Spieler B wird seinen Zug davor deshalb nicht ziehen, da ihm
                                // schon ein besserer Alternativzug mit der Bewertung b bekannt ist
      }
      undoMove();
    }

    if (it is playerAs turn) return a;
    else return b;
}
```

Abbildung 5 Pseudocode Alpha-Beta Suche vgl. [8]

In Abbildung 5 ist der Pseudocode des Alpha-Beta Algorithmus zu sehen. Im Vergleich zum MiniMax Algorithmus kommt beim Pseudocode das Beschneiden des Spielbaums hinzu. Bei einem Beta-Cut wird der Wert a zurückgegeben, was die aufrufende Methode zum Ignorieren des Zugs veranlasst, da es bereits einen Zug mit gleicher oder besserer Bewertung gibt. Man beachte hierbei, dass der zurückgegebene Wert a vom Aufrufer, neben einem Vergleich mit dem aktuell besten Zug, nicht genutzt wird. Es könnte ebenso gut ein noch schlechterer Wert wie zum Beispiel der Wert b oder -∞ zurückgegeben werden. Die aufrufende Methode würde dann ebenfalls den bereits gefundenen, besseren Zug favorisieren.

Das Beschneiden des Spielbaums hat auf den vom Algorithmus ausgewählten Zug keinen Einfluss. So liefert die Alpha-Beta-Suche bei gleichen Eingaben immer denselben Zug wie der MiniMax Algorithmus. Im direkten Vergleich wird die Anzahl besuchter und zu bewertender Blätter bei diesem Algorithmus etwa auf die Quadratwurzel beschränkt [2]. Diese Schätzung ist aber stark davon abhängig, wie früh ein guter Zug in einem Knoten getestet wurde, also wie früh es zu einem Alpha-Cut bzw. Beta-Cut kommt. Wird immer der beste mögliche Zug zuerst getestet, so wird nur noch ein Bruchteil des Suchbaums berücksichtigt. Im Gegensatz zum MiniMax Algorithmus, bei dem die Reihenfolge der zu testenden Züge keine Rolle spielt, lohnt sich bei der Alpha-Beta-Suche also eine Zugvorsortierung. Potenziell gute Züge einer Stellung zu identifizieren um sie früher zu testen ist aber keineswegs trivial - ist doch das eigentliche Ziel des Algorithmus einen solchen Zug zu finden. Im Gegensatz zum Menschen sind für den Computer keine Züge offensichtlich gut oder schlecht. Allerdings werden bei

dem brute-force-artigen Algorithmus hauptsächlich für das Schachspiel untypische Stellungen durchlaufen. Es kann daher davon ausgegangen werden, dass Schlagzüge, welche Figuren des Gegners vom Brett entfernen, in den meisten Fällen korrekt sind und zu dem Verwerfen aller alternativen Varianten führen. Diese sollten daher als erstes getestet werden, was die Laufzeit des Algorithmus bereits stark verkürzt. Eine weitere Idee wird im späteren Kapitel 1.3.3 vorgestellt.

1.2.3. Stärken und Schwächen der Basisalgorithmen

Obwohl die Basisalgorithmen aufgrund ihrer in der Praxis bewiesenen Stärken in praktisch jeder Schach-Engine vertreten sind, weisen sie in ihrer reinen Form einige Schwachstellen auf. Es bleibt dem Geschick des Programmierers überlassen, wie diese reduziert oder vermieden werden können.

Da der *MiniMax* Algorithmus in der *Alpha-Beta-Suche* praktisch enthalten ist und die *Alpha-Beta-Suche* sich nur durch ihre potenziell höhere Geschwindigkeit unterscheidet, gelten die folgenden Betrachtungen für beide Algorithmen:

pro:

- Innerhalb der Suchtiefe wird kein Zug übersehen und damit auch keine komplizierten taktischen Manöver
- Findet nicht nur für Schach sondern ganz universell für alle Nullsummenspiele (Spiele für zwei Personen mit voller Information ohne Zufallselemente) den objektiv besten Zug. Dazu gehören etwa TicTacToe, Mühle, Vier gewinnt, Schach, Dame und GO [9]
- kaum speicherintensiv, müssen doch nur wenig rekursive Funktionsaufrufe zur gleichen Zeit im Speicher gehalten und der Suchbaum sequentiell abgearbeitet werden

contra:

- Während bei TicTacToe der komplette Spielbaum bis zu jeder möglichen Endstellung aufgebaut und so der beste Zug ausgerechnet werden kann, ist die Abarbeitung des gesamten Spielbaumes in Spielen wie Schach und GO aufgrund des hohen Knotengrades und der größeren maximalen Anzahl an Zügen innerhalb eines Spiels nicht möglich. Dem Computer muss hier eine Zeit oder maximale Baumtiefe vorgegeben werden, bis zu welcher der Algorithmus Züge berücksichtigt. Dabei entsteht aber ein neues Problem:

- Die Basisalgorithmen können zwar bis zur maximalen Suchtiefe (dem so genannten Horizont) alle Züge erfassen, ohne dass einer *übersehen* werden kann. Alles was hinter diesem "Horizont" liegt, bleibt aber verborgen. Ist die Suchtiefe eine ungerade Zahl, ist die Engine selbst an den Blättern des Suchbaums am Zug. Kann der Computer direkt am Horizont mit der Dame beispielsweise einen Bauern schlagen, wird die Bewertungsfunktion feststellen, dass diese Variante einen Bauern gewinnen kann. Eigentlich ist der Bauer gedeckt. Wenn der Gegner am Zug ist, wird der Computer die Dame verlieren und deutlich schlechter stehen. Das wird jedoch vom Algorithmus nicht mehr erfasst. Der Computer spielt also übertrieben optimistisch. Handelt es sich bei der Suchtiefe um eine gerade Zahl, ist der Gegner des Algorithmus bei den Blättern am Zug. Kann dieser im letzten Zug noch etwas schlagen, werden diese Varianten als schlecht befunden, auch wenn der Schlagzug direkt widerlegt werden könnte. Der Computer verhält sich dadurch unnötig defensiv. Dieses Problem ist in der Schachprogrammierung sehr populär und wird als "Horizont-Effekt" bezeichnet. Eine interessante Variante des Horizont-Effekts tritt auf, wenn der Computer Materialverlust nicht mehr verhindern kann. Ist beispielsweise ein Turm so angegriffen, dass er auf jeden Fall verloren geht, opfert der Computer evtl. einen Läufer mit einem Schachgebot. Er geht davon aus, dass er nun nur den Läufer verliert, wenn das unvermeidliche Schlagen des Turms nun hinter dem Horizont liegt, also außerhalb der betrachteten Züge. Im nächsten Zug wird er feststellen, dass nach der Verlust des Läufers wiederum der Turm verloren geht und opfert evtl. sogar eine weitere Figur, weil er das Schlagen des Turms dadurch wieder hinter den Horizont verschiebt. Der Computer zieht also de facto die schlechtesten Züge, welche gerade auf dem Schachbrett möglich sind. Die allgemeine Lösung für den Horizont-Effekt liegt in einem eigenen Algorithmus, welcher an den Blättern des Suchbaums die möglichen Schlagfolgen und andere konkrete Bedrohungen auch über den Horizont hinaus verfolgt. Dadurch kann der Horizont-Effekt deutlich abgeschwächt werden. Diese so genannte "Quiescence search" wird im Kapitel 1.3.1 vorgestellt.
- Der Algorithmus arbeitet denkbar langsam, werden doch auch unsinnige Zugfolgen berücksichtigt, welche mehr als 99,999% ausmachen dürften [10]
- Der Algorithmus verlässt sich darauf, dass die Bewertungsfunktion absolut korrekt ist. Das ist aber nicht realisierbar, da es völlig unterschiedliche Schachstellungen gibt die mit einer statischen Bewertung nicht gleichermaßen behandelt werden können. Der Programmierer muss den Computer in unvorhergesehenen Situationen in irgendeiner allgemeinen Form "improvisieren" lassen. Als erster Ansatz können für verschiedene Spielphasen verschiedene Bewertungsfunktionen eingesetzt werden. In der Eröffnungsphase und dem späteren "Mittelspiel" ist beispielsweise die Sicherheit des

Königs von allerhöchster Priorität. Im "Endspiel", welches sich durch sehr wenig Figuren und vor allem Bauern auszeichnet, muss der König als nun verhältnismäßig starke Figur aktiv ins Geschehen eingreifen. Alleine um das zu berücksichtigen sind bereits verschiedene Bewertungsfunktionen nötig.

- In der Eröffnungsphase (bis beide Spieler bereits jeweils 10-20 Züge durchführten) haben sich durch die jahrelangen Erfahrungen im Bereich des Schachs bestimmte *Eröffnungszüge* als die Empfehlenswertesten durchgesetzt. Da eine Schach-Engine den Spielbaum nur bis zu einer gewissen Tiefe durchsucht, was als Taktik interpretiert werden kann, fehlen ihr komplett strategische, d.h. planende Faktoren. Computer spielen gerade in der Eröffnungsphase dadurch seltsam und planlos, ohne dabei aber direkte Fehler zu machen. Die Programmierer versuchen daher die strategischen Schwächen in der Anfangsphase einer Partie durch ein "Eröffnungsbuch" in den Griff zu bekommen, indem gute Eröffnungszüge aus Theoriebüchern gespeichert und während des Spiels abrufbar sind. Als positiver Nebeneffekt benötigen Computer, die Eröffnungsbücher nutzen, für die ersten Züge kaum messbare Zeit.
- Der feste Algorithmus findet bei gleichen Eingaben immer denselben Zug. Selbst wenn zwei Züge die gleiche Bewertung erhalten würden, wird der zweite Zug durch einen Alpha-Cut nicht mehr vollständig ausgewertet. Für die Anwender ist das natürlich langweilig und gewonnene Partien sind beliebig oft reproduzierbar. Das gleiche Problem tritt auch beim Vergleich von zwei Schach-Engines auf. Lässt man diese mehrere hundert Partien gegeneinander spielen, führen alle Partien, in denen die Engines mit denselben Farben starten, zum selben Ergebnis. Auch diesem Problem kann mit Eröffnungsbüchern begegnet werden. Überraschenderweise gibt es in vielen Stellungen mehrere Züge, die absolut gleichwertig sind, aber zu völlig verschiedenen Partien führen. Hier kann gerade bei bekannten Eröffnungszügen eine zufällige Auswahl erfolgen. Darüber hinaus kann die Alpha-Beta-Suche auch so modifiziert werden, dass nach mehreren besten Zügen gesucht wird. Dadurch wird der Baum aber nicht mehr so stark beschnitten, was wiederum zu einer längeren Laufzeit führt.
- Beim sequentiellen Bewerten aller möglichen Züge der Ausgangsstellung findet der Computer unter Zeitdruck evtl. ein Matt in einem Zug nicht, da er die Alpha-Beta Suche mit diesem Zug noch nicht durchgeführt hat, bevor die Zeit abgelaufen ist. Das liegt daran, dass die Zeit, bis zu welcher eine bestimmte Zugtiefe erreicht wird, nicht bekannt ist. Deshalb eignet sich eine vorgegebene Suchtiefe nicht wenn zusätzlich eine Zeitbegrenzung besteht. Gerade bei einer Zeitvorgabe wird daher eine *iterative Tiefensuche* bevorzugt, bei welcher zuerst alle Züge mit einer Suchtiefe von 1 bewertet werden und anschließend die Baumsuche mit immer größer werdenden Suchtiefen

wiederholt wird. Bei Ablauf der Zeit oder Abbruch durch den Benutzer kann so ein der Zeit angemessener qualitativer Zug zurückgegeben werden. Da der Suchbaum jedes Mal neu aufgebaut wird, dauert eine iterative Tiefensuche länger als eine normale Tiefensuche. Die Bewertungen der Züge in der letzten Iteration kann aber als Zugvorsortierung für die nächste Iteration genutzt werden, was zu schnellen Alpha-Cuts führen kann.

1.2.4. Spielstärke von Schach-Engines

Die Spielstärke einer Schach-Engine definiert sich durch die Zugqualität, die diese in einer vorgegebenen Zeit auf einer vorgegebenen Hardware erreichen kann. Da die Spielstärke erfahrungsgemäß mit höheren Suchtiefen steigt, ist es möglich, dass eine sehr einfach geschriebene Schach-Engine mit einem Jahr Bedenkzeit pro Zug eine wesentlich bessere Schach-Engine, der nur eine Sekunde für jeden Zug zur Verfügung steht, schlagen kann. Da es sich beim Aufbau und Abarbeiten des Suchbaums um ein mit der Suchtiefe exponentiell größer werdendes Problem handelt, wird bei verdoppelter Rechenleistung der Hardware nicht immer eine höhere Suchtiefe erreicht. Überhaupt sind die komplexeren Befehle, die von aktuellen Mikroprozessoren unterstützt werden, für die Programmierung von Schachcomputern kaum interessant, da eine Schach-Engine sehr einfache Berechnungen, gewöhnlich sogar ohne Gleitkommazahlen, durchführt und so mit sehr einfachen Programmbefehlen auskommt.

Wir sehen, dass Schach-Engines nur bei gleicher Rechenzeit und gleicher Hardware verglichen werden können. Dabei definiert sich die Spielstärke einer Schach-Engine über mehrere Faktoren:

- Die Geschwindigkeit (in Knoten/Sekunde), in der der Suchbaum aufgebaut und durchlaufen wird. Dies ist abhängig von der internen Darstellung der Daten (vgl. Kapitel 2.2), dem Zuggenerator und einer effizienten Legalitätsprüfung der Züge.
- Die Geschwindigkeit, in der die Zugvorsortierung in allen Knoten stattfindet und deren Qualität. Dies führt zu früheren Alpha- und Beta-Cuts.
- Die Geschwindigkeit, in der die Bewertungen der Endstellungen an den Blättern stattfinden, sowie die Qualität der Bewertungsfunktion.
- Die Treffsicherheit der Auswahl wichtiger Züge, welcher hinter dem Horizont weiter verfolgt werden sollten.

1.3. Fortgeschrittene Algorithmen

Bereits implementierte Basisalgorithmen können durch weitere Algorithmen erweitert werden. Wir werden nun einige der wichtigsten betrachten.

1.3.1. Quiescence search

Die *Quiescence search* (Ruhesuche) beschäftigt sich mit dem in Schach-Suchalgorithmen typischen Horizonteffekt, also dem Beenden der Suche bei einer vorgegebenen Suchtiefe (siehe Kapitel 1.2.3). Solange noch Schlagzüge oder ähnlich interessante taktische Züge auf dem Brett möglich sind, sollten diese mit einem eigenen Algorithmus, einer "*Quiescence search*" tiefer berechnet werden. Dabei wird nach dem Erreichen einer vorgegebenen Suchtiefe eine möglichst kleine Auswahl an Folgezügen untersucht, um keinen zu großen Rechenaufwand zu betreiben. In Abbildung 6 ist nach dem Erreichen des Horizonts bei einer vorher festgelegten Tiefe von 2 (Blätter des blauen Graphen) die weitergehende Suche (rot) in ausgesuchten Blättern angedeutet. Die Folgezüge werden auf ihre taktische Relevanz geprüft und gegebenenfalls weiter verfolgt. Die Aufrufe der Bewertungsfunktionen (schwarze Knoten) werden in diesen Fällen erst an den neuen Blättern durchgeführt. Die gestrichelten, d.h. abgeschnittenen blauen Zweige, wurden in diesem Beispiel durch Alpha-cuts übersprungen, wodurch in diesen Zweigen auch keine Bewertungsfunktionen aufgerufen werden mussten. Beta-Cuts sind in der Grafik nicht dargestellt, da sie erst ab einer Tiefe von 3 möglich sind.

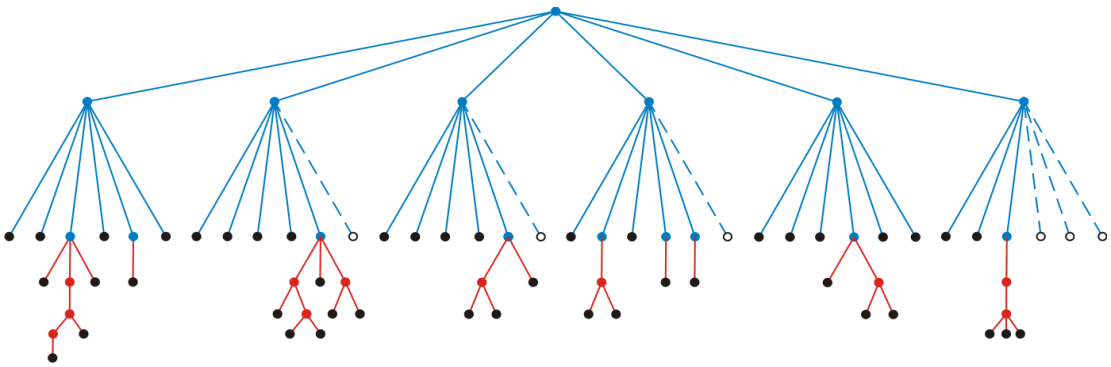


Abbildung 6 Baumsuche mit Alpha-Beta und Quiescence search

Das Ziel der *Quiescence search* ist das Erreichen von "ruhigen Endstellungen" in denen keine taktischen Züge wie Schlag- und Schachgebotszüge mehr möglich sind und die sich damit für die Bewertungsfunktion eignen. Um den Algorithmus terminierend zu formulieren sollte trotzdem eine größere Tiefe als Obergrenze festgelegt werden.

Schon Claude Shannon kategorisierte die Suchalgorithmen in Typ A Algorithmen, wie seinen MiniMax Algorithmus, welche alle Züge berücksichtigen und in Typ B Algorithmen, bei denen nur ausgesuchte Züge betrachtet werden [3]. In letztere Kategorie fällt die Quiescence *search*, welche neben dem Behandeln des Horizont-Effekts auch nach anderen taktischen Manövern sucht.

Das "Beschneiden" des Suchbaums, d.h. das Aussuchen von potenziell wichtigen Zügen, welche nach dem Erreichen des Horizonts weiter verfolgt werden sollen, ist nicht trivial. Es besteht nämlich die Gefahr sehr gute Züge im Suchbaum auszulassen. Die Qualität des Auswahlverfahrens trägt daher einen beträchtlichen Teil zur Spielstärke des Algorithmus bei. Werden beispielsweise nur die zwei potenziell besten Züge jeder Situation berücksichtigt, können dadurch erstaunliche Suchtiefen erreicht werden, für deren vollständige Berechnung aller Baumzweige eigentlich Billionen von Jahren nötig wären.

Dennoch weisen die heutigen Schachcomputer strategische Schwächen auf. Während sie den Menschen bei taktischen Manövern, die innerhalb ihrer Rechentiefe abgeschlossen werden können, überlegen sind, haben Menschen die Möglichkeit, mit langfristig angelegten Manövern zu operieren, deren Ansatz für den Computer im Rahmen seiner Rechentiefe zunächst nicht erkennbar ist. So kann ein Mensch planen, einen noch nicht weit vorgerückten Bauern zur Umwandlung in eine stärkere Figur zu bringen, welcher von Computern aber erst später als Bedrohung erkannt wird.

1.3.2. Hash Tabellen

Ein Schachprogramm durchläuft während der Baumsuche immer wieder identische Schachpositionen, die in unterschiedlicher Reihenfolge der vorangegangenen Züge erreicht werden. Schachspieler sprechen hierbei von "Zugumstellungen". Im Computerschach werden deshalb Hash Tabellen eingesetzt, um den Aufwand der Mehrfachberechnung von Teilbäumen zu verhindern. Dazu muss der Suchalgorithmus lediglich bei jedem Besuch eines Knotens im Spielbaum prüfen, ob die gleiche Schachstellung schon einmal aufgetreten ist und welcher

Wert damals zurückgegeben wurde. Für diesen Knoten müssen dann weder Folgezüge generiert, noch vorsortiert werden. Wir folgen nun weitgehend den Betrachtungen von Stefan Meyer-Kahlen [14], dem deutschen Programmierer der mehrfachen Weltmeister-Engine "Shredder". Dabei betrachten wir aber nur die bei Mediocre eingesetzte Variante:

Um Zugwiederholungen komplett auszuschließen, müsste man alle während einer Suche vorgekommenen Schachpositionen in eine "Zugumstellungstabelle" eintragen. Die Größe dieser Tabelle unterliegt aber den physikalischen Grenzen der Computer. Außerdem müssen die Positionen schnell in der Tabelle gefunden werden können, damit sich der Mehraufwand, welcher durch das Pflegen der Tabelle entsteht, nicht relativiert. Deshalb bedient man sich einer Hash Tabelle. Die Schachpositionen werden dabei in Form einer 64 bit Binärzahl kodiert und abgelegt. Die Hash Tabelle (im Sinne der Schachprogrammierung), in welche diese Zahlen eingetragen werden, ist dabei ein Array mit einer festgelegten Größe. Um eine Schachposition in diesem Array schnell zu finden, wird jeder Stellung ein Hash Key zugeordnet, welcher dem entsprechenden Index der Hash Tabelle entspricht, unter welcher die Schachposition abgelegt wurde. Wird eine neue Schachposition in die Tabelle aufgenommen, wird der 64 bit Wert modulo der Länge der Hashtabelle gerechnet. Das Ergebnis entspricht dem Hash Key, welcher gleichbedeutend mit dem Index ist, an welchem diese Schachposition im Array aufgenommen wird. Natürlich existieren viel mehr Schachpositionen, als eine solche Hashtabelle aufnehmen kann. Deshalb tritt sehr häufig der Fall auf, dass der Hash Key einer Schachposition in der Hashtabelle gesucht und gefunden wird, obwohl die Schachposition sich von der unterscheidet, welche sich in der Hashtabelle bereits unter diesem Index befindet. Es kommt dabei zu einer sogenannten Hashkollision, welche aber leicht daran zu erkennen ist, dass sich die in 64 bit kodierte Schachposition von der in der Hashtabelle unterscheidet. Mit Hashkollisionen wird sehr unterschiedlich verfahren und es ist den Programmierern überlassen, ob die Tabelle die neue Position ignoriert oder die alte überschreibt. In beiden Fällen müssen erneut auftretende Stellungen, welche in der Hashtabelle nicht aufgeführt sind, neu berechnet werden.

Es können allerdings viel mehr Positionen auf einem Schachbrett aufgebaut werden, als mit 64 bit dargestellt werden können. Viele verschiedene Schachstellungen erhalten daher den gleiche 64 bit Kodierung (im folgenden Stellungscode genannt). Wurde eine Stellungsbewertung zusammen mit dem Stellungscode gespeichert und in einer anderen Variante wird in einer eigentlich ganz anderen Stellung der gleiche Stellungscode genutzt, spricht man ebenfalls von Hashkollisionen, welche sich aber von den oben erwähnten unterscheiden. Die Wahrscheinlichkeit, dass eine Hash Kollision aufgrund doppelter

Stellungscodes auftritt ist verschwindend gering. Dieser Fall könnte aber auftreten und dadurch könnte es auch passieren, dass der Computer in einer Stellung deutlich besser steht aber eine später auftretende, sehr schlechte Stellung, durch Nutzung desselben Stellungscodes, dieselbe Bewertung erhält.

Das System, mit dem Schachstellungen auf Stellungscodes abgebildet werden, sollte auch so gestaltet sein, dass Besonderheiten der aktuellen Schachstellung in den Stellungscode einfließen. Dabei sollte die Priorität darauf liegen, dass Stellungen, die sich sehr ähneln, zwingend andere Stellungscodes bekommen. Beim Durchlaufen eines Spielbaums können gewöhnlich auch nach vielen Zügen Gemeinsamkeiten zwischen den besuchten Schachpositionen festgestellt werden. Es sei hierbei nur die Positionen der Bauern genannt, die sich in einer gewöhnlichen Schachpartie nur gemächlich ändern oder andere Figuren, die in der entsprechenden Variante nicht gezogen oder geschlagen wurden. Durch die beschriebene Wahl der Stellungscode-Generierung sind Kollisionen in aktuell möglichen Suchtiefen sogar nahezu ausgeschlossen. Die theoretische Möglichkeit, dass doch einmal eine Hash Kollision dieser Art auftritt und sich dann auch ein schlechterer Zug bis zur Wurzel des Spielbaums durchsetzt, wird von den Schachprogrammierern ignoriert.

In Stellungen mit sehr wenigen Figuren, kommt es zu einem sehr hohen Aufkommen an Zugumstellungen, da in den folgenden Zügen nur eine übersichtliche Zahl an unterschiedlichen Stellungen entstehen kann. Dort machen sich Hash Tabellen extrem bemerkbar: Suchtiefen, die erst nach Tagen erreicht würden, werden nun in Sekunden abgearbeitet. In Stellungen mit vielen Figuren wirken sich Hash Tabellen nicht so stark aus. Der Gewinn an Rechenzeit ist aber auch hier höher, als der für die Hash Verwaltung betriebene Aufwand.

Die maximale Größe der Hashtabelle sollte so gewählt werden, dass keine Auslagerung auf die langsamere Festplatte stattfindet, damit die Engine nicht "ausgebremst" wird. Steht der Engine allerdings nur sehr wenig Zeit für jeden Zug zur Verfügung, wie es in so genannten "Blitzpartien" üblich ist, wird die Hashtabelle zum großen Teil nicht gefüllt sein, bis die Engine sich für einen Zug entscheiden muss. Der Computer erhält also durch großen Arbeitsspeicher in solchen Fällen keinen großen Vorteil.

1.3.3. Killer Heuristik

Eine gute Zugvorsortierung kann die Alpha-Beta Suche sehr stark beschleunigen. Wie im Kapitel "Alpha-Beta-Suche" beschrieben, ist es keine leichte Aufgabe gute Züge zu identifizieren. Die Killer Heuristik ist eine Zugsortierungstechnik, die sich mit bereits bekannten guten Zügen beschäftigt. Verursachte beispielsweise der Zug Springer von e2 nach f4 (kurz Sf4) in einer Tiefe von drei einen Beta-Cut, kann davon ausgegangen werden, dass dies für diese Variante ein starker Zug gewesen ist. Der Zug Sf4 wird deshalb als "Killer Zug" in eine Tabelle eingetragen, die für jede Suchtiefe getrennt starke Züge sammelt. Wenn in einem anderen Baumzweig wieder die Tiefe drei erreicht wird, testet die Killer Heuristik, ob der Zug Sf4 auch hier existiert und legal spielbar ist. Falls das zutrifft, wird der Zug direkt nach den Schlagzügen einsortiert um im Idealfall auch hier einen frühen Beta-Cut zu erreichen. Man beachte dabei, dass Schlagzüge nicht als Killerzüge aufgenommen werden, da diese generell als gute Züge in Frage kommen und noch vor den Killerzügen getestet werden sollten.

Gerade in niedrigen Suchtiefen unterscheiden sich die Spielsituationen nur wenig von der Ausgangsstellung, wodurch die Wahrscheinlichkeit steigt, dass ein "Killer Zug" auch in anderen Varianten ein starker Zug ist. Natürlich sind gerade in kleinen Suchtiefen, also in der Nähe der Wurzel, Beschneidungen besonders effektiv. Gewöhnlich werden pro Suchtiefe nicht mehr als 2-4 verschiedene Züge aufgenommen.

Ein verwandtes Verfahren ist die "History Heuristic". Hier werden alle Züge die Cutoffs verursacht haben unabhängig von der Suchtiefe in eine Liste aufgenommen. Für jeden Zug wird gezählt, wie oft dieser zu einem Cutoff geführt hat. Dadurch können die statistisch relevantesten "Killer Züge" nach vorne sortiert werden.

1.4. Aktuelle Schach-Engines

Es existieren verschiedene regelmäßig veröffentlichte Weltranglisten der bedeutendsten Schach-Engines. Um eine solche zu erstellen wird ein Turnier mit allen zu vergleichenden Engines ausgetragen. Zur Erstellung der aktuellen weltweit anerkannten Weltrangliste des schwedischen Schachcomputervereins SSDF wurden im Jahr 2010 über 110000 Einzelpartien ausgespielt um 50 Engines objektiv vergleichen zu können [11]. Erwähnt sei hierbei, dass bei den besten Engines neben kommerziell vertriebenen Programmen auch Freeware und Open Source Software vertreten sind.

Als quasi Weltmeister kann die Engine "Rybka" angesehen werden. Diese wurde von dem russischen Schachgroßmeister Vasik Rajlich entwickelt und führt viele der verfügbaren Weltranglisten an oder zählt zu deren Spitzenprogrammen. Vermutlich enthält Rybka speziell in der Bewertungsfunktion sowie der Zugvorsortierung mehr „theoretisches Schachwissen" als die Konkurrenzprogramme, deren Programmierer selbst keine Schach-Meistertitel tragen.

Als sehr starke Open Source Engine ist die von dem Franzosen Fabien Letouzey entwickelte C++ Engine "Fruit" bekannt geworden. Im Februar 2006 stand Fruit in der Version 2.2.1 vor dem mehrfachen Weltmeister Shredder auf Platz 1. Die Versionen nach 2.3 sind nur noch kommerziell und ohne offenen Quellcode verfügbar. Viele starke Schach-Engines bauen aber noch heute auf Fruit 2.3 auf [12].
Selbst Vasik Rajlich bestätigte in einem Interview, dass er für seine Engine Rybka viele Elemente aus der letzten offenen Fruit Version entnommen hat [13].

2. Die Schachengine Mediocre

Mediocre wurde von dem Schweden Jonatan Pettersson über mehrere Jahre hinweg entwickelt. Begleitend zur Implementierung wurde ein Blog gepflegt, in dem die Entwicklung in Form eines Tutorials dokumentiert wurde. Der Quellcode ist komplett in Java geschrieben und unter Sourceforge.net verfügbar.

2.1. Eingesetzte Technologien

Auflistung der in Mediocre implementierten bekannten Schachtechnologien:

- Alpha-Beta Suche
- Quiescence search
- Late move reduction (LMR), eine Technik die ähnlich zur Quiescence Search taktisch wichtige Züge am Horizont erkennt um diese weiter zu verfolgen
- Killer Heuristik
- History Heuristic
- Transposition tables (Hash Tabellen)
- UCI und "XBoard" Protokoll (zur Verbindung mit einer GUI)
- Iterative Tiefensuche, vgl. Kapitel 1.2.3
- Static Exchange Evaluation (SEE), eine kurze Überprüfung, ob ein Schlagzug vermutlich zu Materialgewinn führen wird. Wenn ein Bauer eine Dame schlägt, ist das potentiell ein sehr guter Schlagzug, egal ob der Bauer anschließend verloren geht oder nicht. Mit dieser Technik können Schlagzüge nach ihrer Erfolgs-wahrscheinlichkeit sortiert werden.
- Principal Variation Search (PVS), eine Beschleunigung der Alpha-Beta Suche, welche in Verbindung mit einer "iterativen Tiefensuche" genutzt werden kann. Die Knoten, die keine Cutoffs erzeugt haben und damit zu wichtigen Varianten gehören, werden gespeichert und, bei Nutzung einer Iterativ größer werdenden Suchtiefe, in der folgenden zu berechnenden Suchtiefe bevorzugt abgearbeitet.
- Aspiration windows, eine Beschleunigung der Alpha-Beta Suche, welche analog zur Principal Variation Search bei Verwendung der iterativen Tiefensuche angewendet werden kann. Durch die Bewertungen der Wurzelzüge, die in der letzten Iteration

gewonnen wurden, ist in etwa bekannt, welche Bewertung die Wurzelzüge in der nächsten Iteration erhalten werden. Die Werte Alpha und Beta werden nahe diesen Werten initialisiert, obwohl noch kein Zug innerhalb dieser Grenzen gefunden worden ist. Dadurch kommt es sehr schnell zu Alpha- und Beta-Cutoffs. War die Prognose falsch und es existiert kein Zug, der in dieses vorgegebene Fenster fällt, war die Suche in dieser Suchtiefe umsonst und muss erneut mit einem größeren "Alpha-Beta Fenster" oder ohne solche Vorgaben durchgeführt werden.

Es folgen gängige Schachtechnologien, die von Mediocre nicht eingesetzt werden:

- Zugvorsortierung: Neben der groben Vorsortierung von Schlagzügen mit der SEE Technik findet bei den Nicht-Schlagzügen keine weitere Zugvorsortierung statt.
- Pondern: Unter Pondern versteht man das "Rechnen auf Gegnerzeit". Nachdem die Schach-Engine ihren Zug an die GUI weitergeleitet hat, führt sie den in der Suche erwarteten Gegnerzug intern aus und beginnt einen Antwortzug zu berechnen. Falls der Gegner sich wirklich für diesen Zug entscheidet, hat die Engine Zeit eingespart und kann entweder sofort antworten oder den Spielbaum noch tiefer erforschen [15].
- Parallelisierung des Suchalgorithmus. Die Baumsuche, wie sie in der Schach-programmierung durchgeführt wird, hat einen sequentiellen Charakter. Je früher Züge komplett berechnet werden, desto schneller kommt es zu den wirkungsvollen Alpha-Cuts. Threads, welche Baumteile untersuchen, die bei iterativer Abarbeitung weggeschnitten worden wären, beschleunigen die Suche nicht. In diesem Bereich existieren allerdings viele verschiedene Ansätze, wie eine parallele Suche stattfinden kann. Dabei ist aber noch kein Standard gefunden worden, welcher sich für die Schachprogrammierer als besonders effektiv herausgestellt hat. Eine gemeinsame Hash Tabelle verhindert beispielsweise das erneute Untersuchen von Knoten, die bereits durch andere Threads vollständig abgearbeitet wurden. Der Speedup ist allerdings nur bei zwei Threads akzeptabel, da diese Vorgehensweise sehr schlecht skaliert. [18]

2.2. Datenmodell

Die Repräsentation des Schachbretts im Programmcode und das Generieren von Zügen sind eng miteinander verbunden. Ein schnellerer Aufbau des Suchbaumes führt zu höheren Suchtiefen. Designentscheidungen an dieser Stelle sind also von weitreichender Bedeutung. Zuerst sollte erwähnt werden, dass beim Ziehen einer Figur Tests durchgeführt werden müssen, welche sich direkt auf die Laufzeit auswirken. Es muss festgestellt werden, ob die Figur nicht das Brett verlässt und ob der Zug nicht illegal ist, da der König im Schach steht. Gewöhnlich wird die Darstellung als eindimensionales Array einem zweidimensionalen vorgezogen. Betrachten wir eine Brettdarstellung, welche als "Mailbox" bezeichnet wird:

```
int mailbox[120] = {
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
     -1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
     -1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
     -1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
     -1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
     -1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
     -1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
     -1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
};
```

Abbildung 7 Mailbox Darstellung eines Schachbretts [16]

In Abbildung 7 sind die eigentlichen Schachbrettfelder mit Zahlen von 0 bis 63 gekennzeichnet. Felder außerhalb des Bretts sind durch den Wert -1 markiert. Wird eine Figur gezogen, muss entsprechend ihrer Gangart ein Wert auf ihren aktuellen Arrayindex addiert oder subtrahiert werden. Der Rand ist so dimensioniert, dass ein Springer nicht über den Rand hinaus gezogen werden kann. Wird der Springer nach rechts aus dem Brett gezogen, befindet er sich anschließend auf einem Randfeld der linken Seite der Darstellung.

Soll eine Figur nun auf einem normalen Integerarray mit 64 Einträgen gezogen werden, kann in dem Mailbox-Array von Abbildung 7 direkt abgefragt werden, ob es sich bei dem Zielfeld um ein Feld auf oder außerhalb des Bretts handelt. Der alleinige Vergleich des im Mailbox-Array ausgelesenen Wertes mit dem Wert -1 ist für die Geschwindigkeit des Algorithmus bereits sehr günstig. [16]

In Mediocre kommt die so genannte 0x88 Präsentation des Schachbretts zum Einsatz. Dabei wird ein eindimensionales Array mit 128 Einträgen in der Programmlogik überraschend als 16x8 Darstellung betrachtet.

7	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
6	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
5	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
4	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
3	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
2	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
1	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**								

Abbildung 8 0x88 Brettrepräsentation eines eindimensionalen Arrays

Wie in Abbildung 8 zu sehen, befindet sich links das Schachbrett und rechts daneben sind Felder, auf denen die Figuren außerhalb des Schachbretts stehen würden. Ähnlich zur Mailboxdarstellung werden Figuren, die über den Rand des Bretts ziehen auf einem Feld der rechten Seite landen. Es ist zu beachten, dass alle Indices der Felder innerhalb des linken Schachbretts in hexadezimaler Darstellung identisch mit den Koordinaten der Felder sind. So ist der Index 41 auf der Koordinate (4/1). Felder, dessen Index in der Einerstelle oder Sechzehnerstelle größer als 7 ist, liegen also außerhalb des Bretts. Diese Koordinaten, welche dann auf der rechten Seite der Arraydarstellung liegen, werden allerdings nicht genutzt. Es werden auch keine Daten aus ihnen gelesen. Die 0x88 Darstellung bedient sich stattdessen einer einzigen Rechenoperation, mit der sofort festgestellt werden kann, ob der Index eines Feldes nur Zahlen von 0 bis 7 enthält, also zum eigentlichen Schachbrett gehört. Sehen wir uns die Indices in Binärdarstellung an. Achtstellige Binärzahlen, welche am höchstwertigsten oder am vierten Bit eine 1 haben, sind in der hexadezimalen Darstellung an der Sechzehner- oder Einerstelle größer als 7. Eine UND-Verknüpfung mit der binären Zahl 10001000 (hexadezimal 0x88, wonach auch die Darstellung benannt wurde) kann das leicht überprüfen und gibt damit bereits an, ob der Index des Felds zum Schachbrett gehört oder auf einem der Felder außerhalb liegt. Dieser Test stellt eine äußerst günstige Operation für den Computer dar, wird doch nur ein Index mit einer einzigen Rechenoperation überprüft und keine Daten, wie bei der Mailboxdarstellung, aus einem Array gelesen. Die 0x88 Brett Repräsentation erfreut sich sehr hoher Beliebtheit. [17]

In jedem Knoten des Suchbaums müssen die möglichen Züge erstellt werden. Dieses "Generieren" der möglichen Züge orientiert sich logischerweise an der gewählten Brettdarstellung. Dabei müssen die Daten, welche zu einem Zug gehören, so gespeichert werden, dass sich der Zug im Laufe der Baumsuche auch wieder rückgängig machen lässt. Gespeichert werden müssen mindestens:

- Art der Figur
- Startfeld der ziehenden Figur
- Zielfeld der ziehenden Figur
- Ist der Zug ein Spezialzug wie eine Rochade oder ein en-passant Zug (Schlagen im Vorübergehen)
- Bestanden vor dem Zug Rochaderechte oder en-passant Möglichkeiten

Die genannten Spezialzüge können durch die alleinige Angabe von Start- und Zielfeld nicht abgedeckt werden. Die Rochaderechte und en-passant Möglichkeiten können für alle verschiedenen Folgezüge einmalig gesichert werden und müssen nicht für jeden Zug einzeln gespeichert werden. Analog zur äußerst starken Open Source Engine Fruit wurde in Mediocre eine Zugdarstellung als Integerwert gewählt. Dabei wurden die 32 Bits eines Integers auf verschiedene Bereiche eingeteilt. Neben den erwähnten Daten taucht in der Zugdarstellung ein Wert zur Sortierung auf. Dieser wird bei Mediocre derzeit aber nicht genutzt.

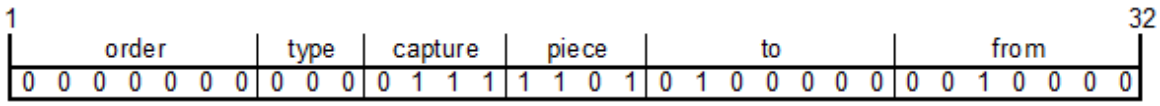


Abbildung 9 Mediocre Schachzug in 32 bit Integer Darstellung

Die Züge als Integer Variablen abzulegen hat verschiedenen Vorteile. Zum einen benötigen die eigentlich umfangreichen Daten sehr wenig Speicher und zum anderen ist es möglich, die Züge in Arrays zu organisieren, welche später überschrieben werden können. Zeitverlust durch das Erstellen von Objekten und deren Verwaltung entfällt hierbei. Die einzelnen Informationen werden in diesem Low-Level Modell nach einer Shift-Operation an die vorgesehene Stelle mit einer Oder-Verknüpfung in die Integer-Variable aufgenommen.

2.3. Stärken und Schwächen

Mediocre hat durch das speicherschonende Datenmodell und die eingesetzten Technologien einen raschen Baumaufbau und erreicht schnell angemessene Suchtiefen. Eine besondere Stärke des Programms ist die flexible Bewertungsfunktion, mit der die Endstellungen in der Baumsuche ihren Wert erhalten. In der Bewertungsfunktion wird eine Stellung mit wenigen verschachtelten if-Abfragen in eine Eröffnungsstellung, eine Mittelspielstellung oder eine Endspielstellung einkategorisiert. Speziell für das Endspiel wurden umfangreiche Vorabfragen implementiert, anhand welcher der Computer entscheiden kann, ob das Spiel aufgrund der Konstellation gewonnen, remis oder verloren ist. Neben der Differenz der Figuren, welche den Parteien zur Verfügung stehen, sind die Stellung der Bauern, die Beweglichkeit der Figuren sowie die Königssicherheit weitere Bewertungskriterien. "Unecht gefesselte" Figuren, welche zwar legal ziehen können, was aber den Verlust einer hinter ihnen stehenden Figur verursachen würde, fließen auch in die umfangreiche Analyse der Stellung ein.

Nachdem ein Matt in Tiefe 3 gefunden wurde, sucht die Iterative Tiefensuche auch die nächst größeren Tiefen ab. Wird mit einem anderen "Startzug" ein Matt in 5 gefunden, ist die Engine auch mit diesem zufrieden und zieht nicht den vorher gefundenen Zug, welcher zum schnelleren Gewinn geführt hätte. Dies scheint bei der Implementierung der iterativen Tiefensuche nicht aufgefallen zu sein. Dieser Bug wirkt sich aber nicht sonderlich aus, da die Engine zu dem Zeitpunkt ohnehin in allen favorisierten Varianten gewonnen hat.

In einem vom Autor dieser Ausarbeitung durchgeführten Testspiel zwischen Mediocre und einem kommerziellen Spitzenprogramm stellten sich massive taktische Schwächen heraus. Die Schachprogramme der Oberklasse können besser unterscheiden, welche Varianten weiter untersucht werden müssen und haben Bewertungsfunktionen, die jahrelang weiter entwickelt wurden. Leider sind die kommerziellen Programme gewöhnlich nicht Open Source oder frei veränderbar und konnten daher nicht als Basis für Bughouse genutzt werden. Darüber hinaus können aus diesen Programmen keine Informationen über deren Implementierung entnommen werden, da dieses Wissen nicht der Allgemeinheit zur Verfügung steht.

3. Die Mediocre Bughouse Erweiterung

Der praktische Teil dieser Bachelorarbeit bestand im Umschreiben von Mediocre in eine Bughouse-Engine. Im Kapitel 3.1 werden zunächst die Unterschiede zwischen der Programmierung einer Schach-Engine und einer Bughouse-Engine erläutert. Im nächsten Abschnitt wird die Entwicklung der neuen GUI beschrieben, welche die zwei Schachbretter verwaltet und die Kommunikation zu allen an der Partie teilnehmenden Engines herstellen muss. Zuletzt wird auf die Modifikation des ursprünglichen Mediocre Quelltextes eingegangen.

3.1. Bughouse Programmierung

Schach-Engines haben in der Vergangenheit lange dazu tendiert zu "gierig" zu sein und Gewinn von "Material" (Schachfiguren) anderen Zügen vorzuziehen. Starke Schachspieler können das ausnutzen, indem sie im Gegenzug zum Verlust von Figuren mehr Bewegungsfreiheit als ihre Gegner erhalten oder andere stellungsspezifische Vorteile anstreben. Man sagt, der Spieler der nun weniger Figuren hat, besitzt im Gegenzug eine gewisse "Kompensation". Aktuelle Schach-Engines und so auch *Mediocre* beziehen in ihre Bewertungsfunktion solche Aspekte mit ein und verzichten ggf. auf Materialgewinn. Im Spiel Bughouse ist der Gewinn von Material allerdings viel höher einzustufen. Wird beispielsweise ein Springer ohne eigenen Materialverlust geschlagen, hat nicht nur der Gegner Materialnachteil an diesem Brett sondern auch dessen Partner am anderen. Der geschlagene Springer wird ja an den Partner weitergereicht, wobei dessen Gegenüber keinen Springer erhält. Das Verlieren von Figuren ist bei Bughouse also im allgemeinen nicht durch Kompensation an nur einem der Bretter auszugleichen. Im Gegensatz dazu ist bei einer Kombination, die zum Matt führt (analog zum normalen Schach), gegen starke Figurenverluste nichts einzuwenden.

Ein großer Unterschied zum normalen Schach ist das Mattsetzen. Steht ein Spieler nach normalen Schachregeln im Matt und hat damit verloren, kann im Bughouse oft noch eine Figur zwischen König und Angreifer "von der Hand" eingesetzt werden. Selbst wenn der Spieler keine Figuren auf der Hand hat, kann der Partner innerhalb der vorgegebenen Zeitbegrenzung noch eine Figur schlagen, welche dann das Matt wieder aufhebt.

Wird ein Bauer entsprechend der normalen Schachregeln, auf der gegnerischen Grundlinie, in eine neue Figur umgewandelt, darf sich der Spieler aussuchen, welche Figur der Bauer darstellt. Wird die Figur (meistens eine Dame) später wieder geschlagen, wird sie bei Bughouse als der ursprüngliche Bauer an den Partner weitergereicht. Die Bughouse-Engine muss beachten, dass ein Tausch von einer regulären Dame mit einer Dame, die für den Partner dann nur ein Bauer ist, am Nebenbrett ein beträchtliches Ungleichgewicht erzeugt.

Eine GUI kommuniziert gewöhnlich mit einer einzigen Engine, welche gegen den Anwender spielt oder Partien für diesen analysiert. Es kommt auch vor, dass die Anwender zwei Engines zu Vergleichszwecken gegeneinander spielen lassen, was als Engineturnier bezeichnet wird. Bei einer Implementierung von Bughouse können bis zu vier Engines, die von der GUI verwaltet werden müssen, parallel eine Partie austragen.

3.2. Erstellen einer GUI mit Programmlogik

Um die Schachbretter einschließlich der Figuren auf und neben den Brettern anzuzeigen, sowie die Züge eingeben zu können, wurde eine Java-Swing GUI entwickelt. Dabei sind die einzelnen Instanzen der Engines in eigenen Threads untergebracht, um ein paralleles Spielen von mehreren Computerspielern zu ermöglichen. Spielen zwei Engines an einem Brett gegeneinander, fiel die Entscheidung, nur eine Engine-Instanz zu nutzen, um die Züge von beiden Spielern abwechselnd zu simulieren. Sollte aber das bei Mediocre nicht verwendete Pondern (Rechnen auf Gegnerzeit) eingesetzt werden, muss natürlich für jeden Computerspieler eine eigene Engine existieren, damit die parallelen Berechnungen von Mehrprozessorsystemen profitieren können. Da derzeit noch kein Pondern implementiert ist, sind allerdings nie mehr als zwei Engines gleichzeitig mit Berechnungen beschäftigt, so dass nur zwei Engine-Prozesse parallel abgearbeitet werden.

Neben der grafischen Anzeige der Spielsituation hat eine GUI die Aufgabe, mit der eingesetzten Schachengine, in unserem Fall sogar mit mehren Engines gleichzeitig, zu kommunizieren. Dabei wurde das aktuelle UCI Protokoll verwendet, welches von Mediocre bereits unterstützt wird. Die GUI muss allerdings selbst entscheiden, ob ein Spieler bereits Matt und die Bughousepartie damit beendet ist. Dafür musste die unter Kapitel 3.1 beschriebene Eigenart der Bughouseregeln beachtet werden, dass ein Spieler erst dann

durch Matt verloren hat, wenn das Matt nicht mehr durch eine neu auf das Brett gestellte Figur aufgehoben werden kann.

Die Übertragung der auszurechnenden Brettsituation von der GUI zur Engine erfolgt im gewählten UCI Protokoll durch Angabe des "FEN-Strings". Ein FEN-String (Forsyth-Edwards-Notation) ist eine sehr kurze textliche Schreibweise einer Spielsituation im Schach. Neben der Stellung der Figuren werden, durch Leerzeichen getrennt, weitere Informationen angehängt. Dazu gehören der Spieler, der am Zug ist, die Rochade- und en-passant Rechte, die Anzahl der gespielten Züge insgesamt und die Anzahl der Züge, welche für die so genannte "50 Züge Regel" relevant sind. Diese Notation musste nun auch Informationen über zusätzliche Figuren enthalten, welche von den Spielern eingesetzt werden können. Dafür wurde der FEN-String auf der GUI- und Engineseite um zwei weitere Felder ergänzt, mit denen Anzahl und Art der geschlagenen Figuren übertragen werden, welche den beiden Spielern zur Verfügung stehen. Die beiden Felder "2P1R" und "-" geben beispielsweise an, dass der weiße Spieler zwei Bauern (Pawns) und einen Turm (Rook) zusätzlich besitzt und der schwarze Spieler keine Figur zum Einsetzen hat. Bei der Übertragung des ausgewählten Zuges, welcher in Textform normal nur aus Start- und Zielfeld Koordinaten besteht, wie der Zug "e2e4", wurde für Bughousezüge das At-Zeichen gewählt, um diese als solche zu kennzeichnen. Wird ein Bauer vom Figurenvorrat auf das Feld e4 eingesetzt, wird dieser Zug von der Engine als "p@e4" übertragen und von der GUI entsprechend interpretiert.

3.3. Modifikation des Mediocre Programmcodes

Um die Engine an das Spiel Bughouse anzupassen, mussten grundlegende Veränderungen vorgenommen werden. Dafür war ein Umfangreiches Reverse Engineering erforderlich. Es existiert neben der Kommunikation mit der GUI kein Standard, nach welchem eine Schachengine aufgebaut sein muss oder was genau eine Engine intern beinhaltet. Ungewöhnliche Designentscheidungen wurden aber vom Programmierer im Quelltext ausführlich kommentiert.

Durch die Bughouseregeln nun möglichen "Einsetzzüge" werden im Folgenden "Bughousezüge" genannt. Um Bughousezüge wie alle anderen Züge als Integer speichern zu können, wurde die Einteilung der Zuginformationen innerhalb einer Integervariablen modifiziert. Im Reverse Engineering wurde festgestellt, dass die Integereinteilung eines abgespeicherten Schachzugs, wie sie im Kapitel 2.2 beschrieben wurde, zwar einen Wert für

eine Vorsortierung vorsieht, dieser "Order"-Bereich (siehe Abbildung 10) aber im Quelltext nicht genutzt wird. Deshalb konnte ein Bit des Sortierungsfeldes als Flag reserviert werden, um Bughousezüge identifizieren zu können. Der Bereich "type" war bereits für alle Arten von Sonderzügen wie Rochaden, en-passant und die verschiedenen Figurenumwandlungen belegt.

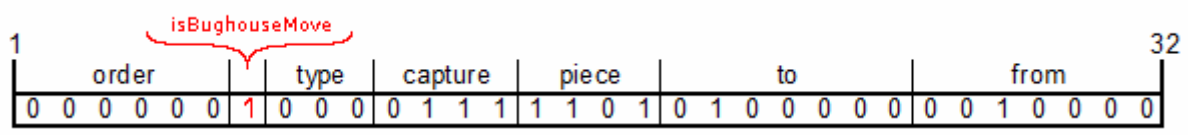


Abbildung 10 Neue Einteilung eines Zugs in 32 Bit Integer Darstellung

Diese Unterscheidung zwischen Standardzügen und Bughousezügen ist sehr wichtig, da Bughousezüge kein Feld besitzen, von der aus die Figur gezogen wurde. An den zahlreichen Stellen im Quelltext, bei denen auf den "from" Bereich (siehe Abbildung 10) eines Zuges zugegriffen wird, musste überprüft werden, ob ein Bughousezug vorliegt und ein dementsprechender Workaround geschaffen werden. Das war nicht nur bei den Methoden doMove() und redoMove() der Suchalgorithmen der Fall, sondern auch bei der Zusammenstellung der Killer-Züge, welche auch Bughousezüge beinhalten können, beim Aktualisieren einer mitgeführten Liste der aktuellen Bauernstellung für Bewertungszwecke, beim Erstellen des Stellungscodes für die Hash Tabelle und vielen anderen Prozeduren.

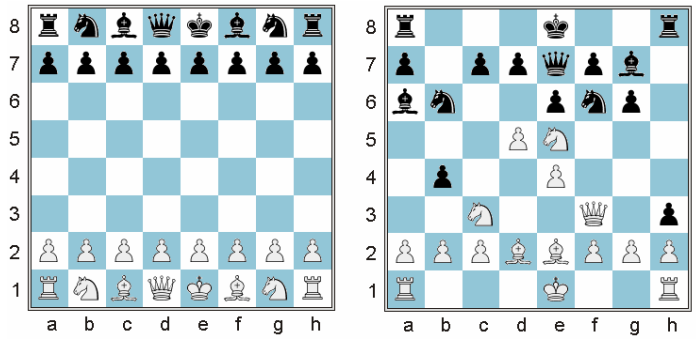
Bei Mediocre werden an verschiedenen Stellen unterschiedliche Arten von Zuglisten erstellt. Diese verschiedenen Zuggeneratoren (siehe Kapitel 1.2.1) sollten nun auch mögliche Bughousezüge generieren. Dafür musste das Datenmodell um die für Bughouse zur Verfügung stehenden Figuren erweitert werden. Die entsprechenden Informationen der Startstellung werden hierbei aus dem Fen-String ausgelesen. Hierbei musste zwischen dem Erstellen der Rootzüge für die iterative Tiefensuche und dem Erstellen von Zuglisten in der rekursiven Alpha-Beta-Suche unterschieden werden. Bei neuen Zuglisten für einen Knoten im Spielbaum werden die Figuren, welche die Züge ausführen, bei Mediocre in Kategorien unterteilt. Es werden "gleitende" Figuren wie Damen, Läufer und Türme von "setzenden" wie Bauern, Springer und Königen unterschieden. Diese Unterteilung wird genutzt, um die Legalitätsprüfung eines Zuges schneller durchführen zu können. Bughousezüge sind keines von beiden, können keine dieser Optimierungen nutzen und müssen auf diese Schnelltests verzichten. Dafür war die Einführung einer dritten Figurenart nötig, damit die Bughousezüge in den Zuglisten überhaupt im Suchalgorithmus beachtet werden.

An der Bewertungsfunktion waren starke Modifikationen nötig, um trotz der neuen Spieleigenheiten sinnvolle Bewertungen vergeben zu können. Die Bewertungsfunktion von Mediocre beschreibt sehr detailliert die Kräfteverhältnisse zwischen den beiden Partien und auch viele Spezialfälle, welche beim Schach auftreten können. Stellungen während einer Bughouse-Partie sind teilweise sehr unterschiedlich zu bewerten. Es erschienen daher u. a. folgende Modifikationen sinnvoll, welche natürlich sehr spielbezogen sind und auf eigenen Erfahrungen mit dem Spiel beruhen:

- Material, welches den Spielern zum Einsetzen zur Verfügung steht, musste in die Materialgegenüberstellung aufgenommen werden.
- Der Besitz eines Läuferpaares bekommt keinen besonderen Bonus mehr, wie es in der Schachtheorie üblich ist. Dieser Quelltext wurde auskommentiert.
- Extrem positive Bewertungen für Bauern, welche kurz vor der Umwandlung stehen und vermutlich nicht mehr aufgehalten werden können, wurden entfernt. Durch das Einsetzen von Figuren nach den Bughouseregeln können Bauern gewöhnlich noch gestoppt werden.
- Es ist nicht gut Türme und Damen (im Schach "Schwerfiguren" genannt) auf das Brett zu setzen, wenn keine konkrete Gewinnchance besteht. Diese Figuren können leicht Angriffen, durch neu eingesetzte Figuren, ausgesetzt sein. Deshalb wurde ein Bonus für alle Schwerfiguren eingeführt, welche vorhanden, aber noch nicht auf das Brett eingesetzt wurden.
- Es ist gut Läufer, Springer und Bauern schnell auf das Brett zu stellen. Sie helfen durch ihre bloße Anwesenheit in Angriff und Verteidigung. Noch nicht eingesetzte Figuren dieser Art bekamen entsprechende Abzüge.
- Eine Prüfung auf eine Remissituation (d.h. die Partie würde in dieser Stellung ein Unentschieden erzwingen) wurde modifiziert. Im Schach sind Stellungen bereits Remis, wenn keine Partei mehr Figuren besitzt, mit welchen der Gegner Matt gesetzt werden kann. Bei Bughouse kann natürlich jederzeit wieder eine neue Figur hinzukommen, welche die Remissituation in eine klare Gewinnstellung für einen Spieler verwandelt. Außerdem ist eine Stellung kaum Remis, wenn die Spieler noch Material zum Einsetzen besitzen. Der Quelltext wurde aber nicht gelöscht, da die Engine so ihre normalen Schachfähigkeiten behält, wenn beiden Partien keine Figuren zum Einsetzten besitzen und das Spiel am Nachbarbrett zum Erliegen gekommen ist.

3.4. Automatische Engine Tests

In der Schachprogrammierung sind Performancetests (kurz Perft) sehr beliebt. Dabei werden alle legalen Züge aus einer Position bis in eine vorgegebene Tiefe gezählt. Dabei werden weder Alpha- noch Beta-Cuts durchgeführt und auch keine Bewertungen an den Blättern. Neben der neutralen Messung der Geschwindigkeit, in welche der Spielbaum mit den verwendeten Datenstrukturen aufgebaut wird können so auch Fehler in der Programmlogik gefunden werden. Im Internet sind zahlreiche Teststellungen mit dazugehörigen Zuganzahlen für verschiedene Suchtiefen verfügbar. Mediocre besitzt für normales Schach bereits seinen eigenen Performancetest und kann aufgrund des schnellen Baumaufbaus in etwa 30 Sekunden alle Züge bis in Tiefe 6 erforschen (siehe Abbildung 11 Ausgangsposition 1). In einer fortgeschrittenen Stellung, in der die Figuren bereits mehr Bewegungsfreiheit besitzen, aber noch keine Figur geschlagen wurde, wird nur noch eine Tiefe von 5 in akzeptabler Zeit erreicht (siehe Abbildung 11 Ausgangsposition 2).



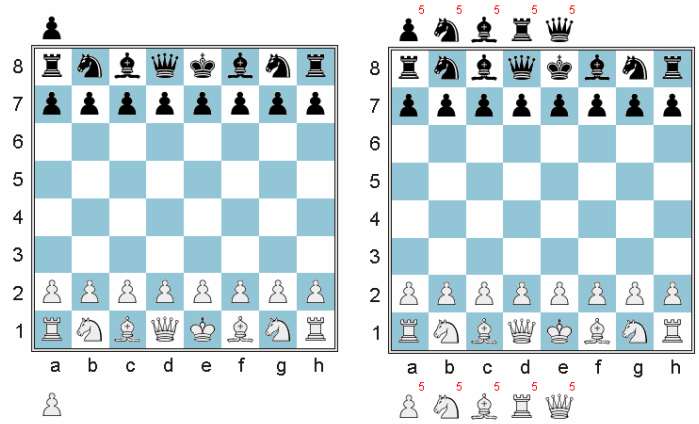
```
Pos 1
  Depth: 1 Answer: 20 (Correct) Time: 0.001
  Depth: 2 Answer: 400 (Correct) Time: 0.004
  Depth: 3 Answer: 8902 (Correct) Time: 0.075
  Depth: 4 Answer: 197281 (Correct) Time: 0.240
  Depth: 5 Answer: 4865609 (Correct) Time: 1.184
  Depth: 6 Answer: 119060324 (Correct) Time: 29.466
Pos 2
  Depth: 1 Answer: 48 (Correct) Time: 0.000
  Depth: 2 Answer: 2039 (Correct) Time: 0.000
  Depth: 3 Answer: 97862 (Correct) Time: 0.024
  Depth: 4 Answer: 4085603 (Correct) Time: 1.106
  Depth: 5 Answer: 193690690 (Correct) Time: 48.246
```

Abbildung 11 Perft von zwei Schach-Ausgangsstellungen

Durch eine Alpha-Beta Suche mit einer anschließenden Quiescence search sind natürlich deutlich größere Suchtiefen erreichbar. Bei diesem Test handelt es sich nur um eine Vergleichbarkeit der Zuggeneratoren in den Bereichen Korrektheit und Geschwindigkeit.

In der Weiterentwicklung zur Bughouseengine wurde auf eine Abwärtskompatiblität zu den bekannten Perft-Zuganzahlen geachtet. So können die Ergebnisse von Teststellungen, in denen die Spieler keine Figuren zum Einsetzen besitzen, mit den bekannten Perft Teststellungen des normalen Schachs bestätigt werden. Für Bughouse sind solche Teststellungen nicht verfügbar und die korrekten Zuganzahlen sind daher nicht bekannt. Da die GUI einen eigenen Zuggenerator besitzt, um die Legalität der vom Anwender eingegebenen Züge zu prüfen, konnten dessen Zuganzahlen mit den Zuggeneratoren der modifizierten Mediocre-Engine verglichen werden. Da die GUI und Mediocre unterschiedliche Datenmodelle besitzen, werden auch die Züge auf verschiedene Arten generiert. Es handelt sich also um einen objektiven Test, die Ergebnisse zu vergleichen. Die verschiedenen Zuggeneratoren ermöglichen dadurch also eine gegenseitige Bestätigung der Perft Zahlen.

In den folgenden zwei Beispielen für Bughouse-Positionen wurde in Abbildung 12 zwei mal die Grundstellung eines Schachspiels als Ausgangsstellung gewählt. Im ersten Fall steht den Spielern je ein Bauer seiner Farbe zusätzlich zur Verfügung. Im ersten Zug kann der Spieler Weiß diesen, statt eines Standardzuges, auf eines der 32 leeren Felder einsetzen. So sind in der Grundstellung statt wie gewöhnlich nur 20 normalen Schachzüge nun 52 verschiedene Züge möglich. Durch den stark erhöhten Knotengrad, im Gegensatz zu den vorangegangenen Schachtests, kann eine Tiefe von 5 erst in 4 Minuten erreicht werden.



```
BugPos 1
  Depth: 1 Answer: 52 (Correct) Time: 0.010
  Depth: 2 Answer: 2658 (Correct) Time: 0.038
  Depth: 3 Answer: 87470 (Correct) Time: 0.538
  Depth: 4 Answer: 2867427 (Correct) Time: 7.776
  Depth: 5 Answer: 87817863 (Correct) Time: 04:04.727
BugPos 2
  Depth: 1 Answer: 180 (Correct) Time: 0.001
  Depth: 2 Answer: 31181 (Correct) Time: 0.076
  Depth: 3 Answer: 5456259 (Correct) Time: 13.748
  Depth: 4 Answer: 927176248 (Correct) Time: 39:23.025
Total time: 43:49.943
```

Abbildung 12 Perft von zwei Bughouse-Stellungen

Ein extremes Beispiel ist in der zweiten untersuchten Stellung von Abbildung 12 zu sehen. Hier stehen beiden Spielern in der Ausgangsstellung eines Schachspiels bereits jeweils 5 Figuren aller 5 Figurenarten (mit Ausnahme von Königen) zur Verfügung. Bereits im ersten Zug kann der weiße Spieler nun 5 verschiedene Figuren auf die 32 leeren Felder einsetzen, womit 160 Züge zu den 20 Standardzügen hinzukommen. Bei dieser, normal nie auftretenden Situation, wird eine Tiefe von 4 erst nach 40 Minuten erreicht. Wenigstens eine Tiefe von 3 steht bereits nach 13 Sekunden zur Verfügung. Im Vergleich zu den gezeigten Schachtests sind Bughousestellungen erwartungsgemäß aufwendiger zu berechnen. Hierbei sind aber Alpha- und Beta Cuts noch nicht berücksichtigt. Wird bei einem Cut eine Berechnung abgebrochen, werden bei Bughouse noch viel mehr Baumzweige als beim normalen Schach übersprungen. Bughouse profitiert dadurch natürlich immens von der Alpha-Beta Suche.

Zusätzlich zu zahlreichen Performanceteststellungen, welche lediglich die Richtigkeit der Zuggeneratoren ansatzweise bestätigen können, wurden verschiedene Schachtests wie beispielsweise "Matt in x Zügen" als JUnit Tests implementiert. Der erwähnte Mediocre-Bug aus Kapitel 2.3 erwies sich hierbei als störend. Wenn in einer Teststellung ein Matt in 4 Zügen gefunden werden soll, muss zwingend die Suchtiefe 4 vorgegeben werden, damit der Computer sich nicht für eine alternative Zugfolge entscheidet, die ein späteres Matt enthält. Dadurch war es nur durch manuelle Tests möglich, die Fähigkeiten der Engine bei einer festen Zeitvorgabe (=dynamische Suchtiefe) bewerten zu können. Abgesehen von den Auswirkungen des Bugs konnten durch die Teststellungen dennoch die taktischen Fähigkeiten der Engine vor und nach den Bughousemodifikationen überprüft werden.

Um die Engine auch hinsichtlich ihrer Bughousefähigkeiten zu testen, wurden zahlreiche Teststellungen mit einsetzbaren Figuren implementiert, welche als Knobelaufgaben für menschliche Schachspieler konzipiert wurden. Darüber hinaus wurde die Engine verschiedenen Bughousefans vorgeführt, welche zwar viele Partien spielten, aber keine gewannen. Der Autor dieser Arbeit konnte sich immerhin an die Spielweise der Engine anpassen, um in der Eröffnung Vorteile zu erhalten. Ein Vergleich mit den Bughouse-Engines, gegen die auf Servern gespielt werden kann, gestaltet sich als schwierig, da diese ihre Spielstärke durch direktes Ziehen, ohne Bedenkzeit zu verbrauchen, erlangen. Subjektiv spielt Mediocre bei 4 Sekunden Bedenkzeit bereits deutlich besser, ohne den Umstand auszunutzen, auch ohne Zeitverlust Züge ausführen zu können.

4. Zusammenfassung

In dieser Ausarbeitung wurden die Basisalgorithmen und einige der fortgeschrittenen Algorithmen der Schachprogrammierung beschrieben. Viele dieser Techniken lassen sich, wie in Kapitel 1.2.3 festgestellt, auch auf andere Spiele übertragen. Hierbei stellte sich die Frage, ob bei der Schachvariation Bughouse, trotz der viel höheren Anzahl möglicher Züge, eine akzeptable Spielstärke erreicht werden kann.

Mediocre stellt für eine Bughouse-Engine eine gute Basis dar, denn viele der modernen Schachprogrammiertechniken sind bereits vorhanden. Das Datenmodell lässt bereits schnelle Zuggenerierungen zu und ein großer Teil der Bewertungsfunktion eignet sich auch für Bughousestellungen.

Bei der Implementierung von Bughouse zerfiel die Entwicklung in die beiden getrennten Bereiche der GUI Programmierung und der Engine-Entwicklung. Dabei mussten die Eigenheiten der veränderten Spielregeln beachtet werden.

Die GUI wurde analog zur Mediocre-Engine in Java geschrieben. Da bei Bughouse mehrere Computergegner gleichzeitig an einer Bughousepartie teilnehmen können, werden Mehrprozessorsysteme automatisch unterstützt.

Die Programmierung der Bughouse-Engine auf Basis von Mediocre fand in einzelnen getrennten Schritten statt. Dazu gehörte das Integrieren von Bughousezügen in das bei Mediocre implementierte Datenmodell, sowie das Generieren von Bughousezügen in den abzuarbeitenden Zuglisten. Die Bewertungsfunktion bedurfte sehr umfangreicher Änderungen, da die Bughouseregeln andere Ansichten über gegebene Spielsituationen mit sich bringen.

Ein eigentlich zu vernachlässigender Bug in der Implementierung von Mediocre erschwerte die automatischen Tests. Dennoch konnten zahlreiche Teststellungen, welche von diesem Bug nicht betroffen waren, von der neuen Bughouse-Engine korrekt abgearbeitet werden, was hohe taktische Fähigkeiten bestätigt. In Partien gegen die Engine wirkt sich der Bug weder bei Bughouse noch im normalem Schach aus.

Die entwickelte Bughouseengine erreicht bereits in einer vorgegebenen Zeit von etwa 4 Sekunden eine beträchtliche Spielstärke. Diese lässt sich noch weiter steigern. Dadurch kann die Engine in weniger Zeit die gleiche Zugqualität erreichen oder bei gleicher Bedenkzeit einen noch stärkeren Spielpartner darstellen. Die Besonderheit von Bughouse, die Nachbarpartie beobachten zu können, kann der Engine eine Prognose geben, welche Figurenarten in den nächsten Zügen voraussichtlich geschlagen werden und der Engine zur Verfügung stehen. Eine direkte Kommunikation zwischen zwei Engines, welche im Team spielen, oder auch zwischen einer Engine und einem menschlichen Partner, ist auch denkbar. Hier kann die Engine eine Figurenart, welche sie nicht hat, auf taktische Einsetzmöglichkeiten testen. Daraufhin kann, ähnlich wie eine Partie zwischen menschlichen Teams, eine Anfrage stattfinden, welche die gewünschte Figurenart und die daraus resultierenden Erfolgsaussichten enthält. Kann beispielsweise mit einem Springer direkt matt gesetzt werden, sollte der Partner an seinem Brett versuchen solch eine Figur zu schlagen, solange die Mattmöglichkeit am Nebenbrett noch besteht.

Die Eröffnungsphase würde sehr von einem für Bughouse geschriebenen Eröffnungsbuch profitieren. Leider gibt es wenig Literatur darüber, welche die besten Eröffnungszüge sein könnten. Um die reine Spielstärke der Engine zu steigern, sollte weitere Entwicklungszeit in die Bewertungsfunktion investiert werden. Dabei können noch viele neue Bewertungskriterien aufgenommen werden. Dabei ist aber darauf zu achten, dass der hinzukommende Mehraufwand die Suche nicht zu sehr verlangsamt, da sich die bessere Qualität der Zugbewertungen mit kleineren Rechentiefen relativieren würde.

Die Engine kann durchschnittliche Schachspieler, welche Bughouse selten spielen, schon jetzt regelmäßig schlagen. Es ist sogar schwer die Spielstärke zu drosseln, um Anfängern eine Chance zu geben. Bei einer Suchtiefe von 1 werden durch die weiterführenden Suchvorgänge bereits viele Angriffe auf die Engine erfolgreich abgewehrt und Chancen wahrgenommen. Es kann also ein positives Fazit gezogen werden, denn die Arbeit hat gezeigt, dass die bekannten Schachprogrammierungstechnologien auch auf Spiele wie Bughouse, trotz des höheren Knotengrads, erfolgreich angewendet werden können. Es ist sogar noch großes Potenzial vorhanden die Spielstärke weiter zu steigern, um auch für sehr starke Bughousespieler ein ebenbürtiger Gegner zu sein.

5. Literaturverzeichnis

[1] Wikipedia: Künstliche Intelligenz, http://de.wikipedia.org/wiki/K%C3%BCnstliche_Intelligenz

[2] H. P. Ketterling, Shannons Ideen zum Computerschach, Erschienen in der Zeitschrift "Rochade" Ausgabe 3 2010, S. 62 ff.

[3] Claude Shannon (1949), Programming a Computer for Playing Chess, http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf

[4] http://upload.wikimedia.org/wikipedia/commons/6/6f/Minimax.svg

[5] Chess Programming WIKI, Negamax Search, http://chessprogramming.wikispaces.com/Negamax

[6] Chess Programming WIKI, Alpha-Beta Search, http://chessprogramming.wikispaces.com/Alpha-Beta

[7] Alpha Beta Suche, Wikipedia, http://upload.wikimedia.org/wikipedia/de/8/83/Alpha_beta.png

[8] B Monien, U. Lorenz, D. Warner, Der Alphabeta-algorithmus für Spielbaumsuche, http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo19.php

[9] Minimax-Algorithmus, http://www.uni-protokolle.de/Lexikon/Minmax-Algorithmus.html

[10] W. Kramnik, J. Schmieder (2006), Interview zum Turnier "Kramnik gegen Deep Fritz, http://www.sueddeutsche.de/sport/kramnik-gegen-deep-fritz-einem-computer-von-aldi-wuerde-ein-solcher-fehler-nicht-unterlaufen-1.736147

[11] Svenska schackdatorföreningen (SSDF), Weltrangliste eines schwedischen Schachcomputervereins, http://ssdf.bosjo.net/list.htm

[12] Chess Programming WIKI, Fruit Engine, http://chessprogramming.wikispaces.com/Fruit

[13] Chess Programming WIKI, Rybka Engine, http://chessprogramming.wikispaces.com/Rybka

[14] Stefan Meyer-Kahlen (2000), Was sind eigentlich - Hash Tabellen, http://www.scleinzell.schachvereine.de/p_basiswissen/hash.shtml

[15] Computer & Schach, was spielt sich da ab?, Alfred Pfeiffer, http://www-user.tu-chemnitz.de/~apf/Informatik-Spielend/pcschach.html

[16] TSCP Engine, http://www.tckerrigan.com/Chess/TSCP

[17] Jonatan Pettersson (2007), The 0x88 representation, http://mediocrechess.varten.org/guides/the0x88representation.html

[18] Chess Programming WIKI, Parallel Search, http://chessprogramming.wikispaces.com/Parallel+Search

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Hofgeismar, den 10.05.2011

_____

Nikolas Luke